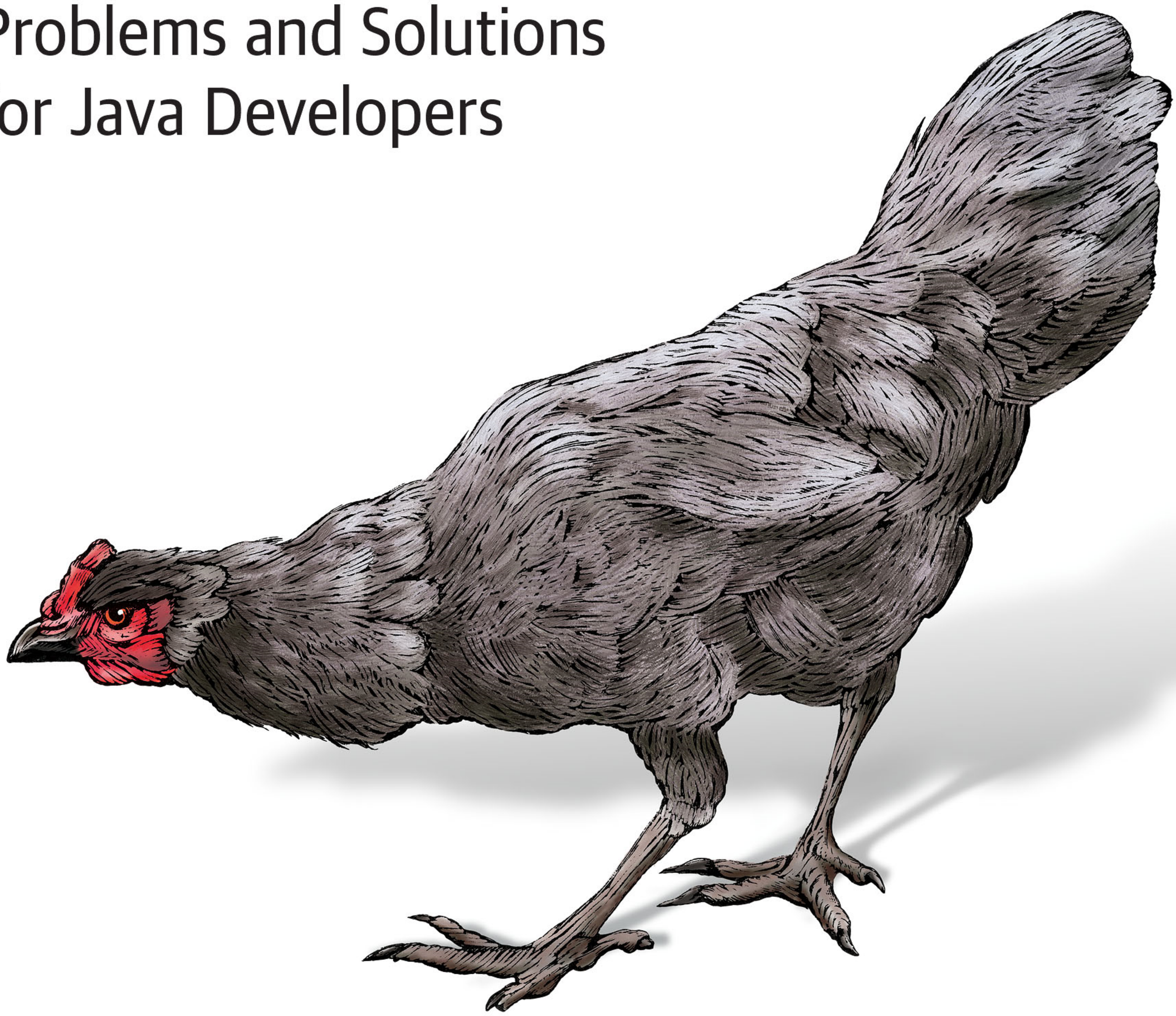
O'REILLY®

5th Edition

Java Cookbook

Problems and Solutions for Java Developers



Ian F. Darwin

FIFTH EDITION

Java Cookbook

Problems and Solutions for Java Developers

Ian F. Darwin

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Java Cookbook
by Ian F. Darwin

Copyright © 2025 RejmiNet Group, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Brian Guerin
Development Editor: Angela Rufino
Production Editor: Beth Kelly
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: Kim Cofer

Indexer: Potomac Indexing, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

June 2001: First Edition
June 2004: Second Edition
July 2014: Third Edition
March 2020: Fourth Edition
February 2025: Fifth Edition

Revision History for the Fifth Edition
2025-02-06: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098169978* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Java Cookbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-16997-8

[LSI]

To everyone who cares about Java, one of the best programming languages.

To my dear wife Betty.

To our son Benjamin and daughter Margaret.

In Memoriam to our son

Andrej Cerar Darwin 1989–2014

Friend, writer, dreamer, gamer, volunteer.

Table of Contents

Preface ..... **xiii**

1. Getting Started: Compiling and Running Java **1**
1.0 Introduction 1
1.1 Hello, World: Compiling and Running Java with the Standard JDK 2
1.2 Hello, World of Classless Main 21P 4
1.3 Downloading and Using the Code Examples 7
1.4 Compiling, Running, and Testing with an IDE 12
1.5 Exploring Java with JShell 11 19
1.6 Using CLASSPATH Effectively 21
1.7 Documenting Classes with Javadoc 23
1.8 Beyond Javadoc: Annotations/Metadata 30
1.9 Packaging and Running JAR Files 32
1.10 Creating a JAR That Supports Multiple Versions of Java 35
1.11 Packaging Web Tier Components into a WAR File 37
1.12 Compiling and Running Java: GraalVM for Better Performance 39
1.13 Getting Information About the Environment, OS, and Runtime 41

2. Software Development, Testing, and Maintenance **49**
2.0 Introduction 49
2.1 Designing Applications: Packages, Modules 49
2.2 Using the Java Modules System 52
2.3 Using JPMS to Create a Module 56
2.4 Automating Compilation, Testing, and Deployment with Apache Maven 61
2.5 Automating Compilation, Testing, and Deployment with Gradle 66
2.6 Automating Dependency Management with Maven and Gradle 68
2.7 Dealing with Deprecation Warnings 73
2.8 Batch Refactoring for Warnings and Migrations 75
2.9 Maintaining Code Correctness with Unit Testing: JUnit 77

2.10 Isolating the Test Target with Mock Objects and Mockito 81
2.11 Logging: Network or Local 82
2.12 Setting Up SLF4J 84
2.13 Network Logging with Log4j 86
2.14 Network Logging with java.util.logging 91
2.15 Maintaining Your Code with Continuous Integration 95
2.16 Performance Timing 101
2.17 Creating a Custom JDK Distribution with jlink 104
2.18 Creating Platform-Specific Installers with jpackage 106

3. Strings and Things. 109
3.0 Introduction 109
3.1 Taking Strings Apart with Substrings, Tokenizing, and Trimming Methods 112
3.2 String Formatting with Formatter and printf() 115
3.3 Building Strings with StringBuilder 121
3.4 Processing a String One Character at a Time 124
3.5 Aligning, Indenting, and Unindenting Strings 126
3.6 Converting Between Unicode Characters and Strings 129
3.7 Reversing a String by Word or by Character 132
3.8 Expanding and Compressing Tabs 133
3.9 Controlling Case 138
3.10 Adding Nonprintable Characters into a String 140
3.11 Creating a Message to the World with I18N Resources 141
3.12 Using a Particular Locale 143
3.13 Creating a Resource Bundle 145
3.14 Program: A Simple Text Formatter 146

4. String Matching with Regular Expressions. 147
4.0 Introduction 147
4.1 Regular Expression Syntax 149
4.2 Checking If a String Matches a Regex 156
4.3 Grouping: Specifying Parts of the Regex 160
4.4 Finding the Matching Text 162
4.5 Replacing the Matched Text 165
4.6 Printing All Occurrences of a Pattern 168
4.7 Controlling Case in Regular Expressions 171
4.8 Matching Accented, or Composite, Characters 172
4.9 Matching Newlines in Text 173
4.10 Program: Full Grep 175

5. Numbers. 177
5.0 Introduction 177
5.1 Checking Whether a String Is a Valid Number 180

5.2 Converting Numbers to Objects and Vice Versa 182
5.3 Taking a Fraction of an Integer Without Using Floating Point 183
5.4 Working with Floating-Point Numbers 184
5.5 Formatting Numbers 189
5.6 Converting Among Binary, Octal, Decimal, and Hexadecimal 194
5.7 Operating on a Range of Integers 195
5.8 Formatting with Correct Plurals 197
5.9 Generating Random Numbers 199
5.10 Multiplying Matrices 201
5.11 Optimizing Large Arithmetic Operations with Vector Operations 22C 203
5.12 Using Complex Numbers 207
5.13 Handling Very Large Numbers 210
5.14 Program: TempConverter 212

6. Dates and Times. 215
6.0 Introduction 215
6.1 Finding Today's Date 218
6.2 Formatting Dates and Times 220
6.3 Converting Among Dates/Times and Epoch Seconds 225
6.4 Parsing Strings into Dates 226
6.5 Difference Between Two Dates 229
6.6 Adding to or Subtracting from a Date 231
6.7 Calculating Recurring Events 232
6.8 Computing Dates Involving Time Zones 234
6.9 Interfacing with Legacy Date and Calendar Classes 235

7. Structuring Data with Java. 239
7.0 Introduction 239
7.1 Using Arrays for Data Structuring 240
7.2 Resizing an Array 242
7.3 Simplifying Array Handling with the Arrays Class 243
7.4 The Collections Framework 246
7.5 Lists: Like an Array, but More Dynamic 247
7.6 Using Generic Types in Your Own Class: Stack Demo 252
7.7 How Shall I Iterate Thee? Let Me Enumerate the Ways 256
7.8 Avoiding Duplicate Values with a Set 259
7.9 Mapping with Hashtable and HashMap 261
7.10 Storing Strings in Properties and Preferences 263
7.11 Sorting a Collection 268
7.12 Finding an Object in a Collection 274
7.13 Converting Between Collections and Arrays 276
7.14 Making Your Own Data Structures Iterable 277
7.15 Multidimensional Structures 280

8. Object-Oriented Techniques. 283
8.0 Introduction 283
8.1 Object Methods: Formatting Objects with toString(), Comparing with Equals 286
8.2 Constructor Simplification: Statements Before super(…) 22P 294
8.3 Using Inner Classes 295
8.4 Simplifying Data Objects with Records (or Lombok) 297
8.5 Providing Callbacks via Interfaces 300
8.6 Polymorphism/Abstract Methods 304
8.7 Improving Interfaces with Default, Static, and Private Methods 305
8.8 Using Typesafe Enumerations 307
8.9 Using Type Pattern Matching 311
8.10 Avoiding NPEs with "Optional" 313
8.11 Controlling Subclassing with Sealed Types 17 315
8.12 Enforcing the Singleton Pattern 317
8.13 Roll Your Own Exceptions 320
8.14 Using Dependency Injection 321
8.15 Combining Java Features for Data-Oriented Programming 324

9. Functional Programming Techniques: Functional Interfaces, Streams, and Parallel Collections. 327
9.0 Introduction 327
9.1 Using Lambdas/Closures Instead of Inner Classes 330
9.2 Using Predefined Lambda Interfaces or Rolling Your Own 335
9.3 Simplifying Processing with Streams 338
9.4 Simplifying Streams with Collectors 339
9.5 Simplifying Streams with Stream Gatherers 22P 343
9.6 Simplifying Streams with Your Own Stream Gatherer 22P 344
9.7 Improving Throughput with Parallel Streams and Collections 346
9.8 Using Existing Code as Functional with Method References 348
9.9 Java Mixins: Mixing in Methods 352
9.10 Functional Programming with Flow and Reactive Streams 354

10. Input and Output: Reading, Writing, and Directory Tricks. 357
10.0 Introduction 357
10.1 Discovering Filesystem Paths 360
10.2 Getting and Setting File and Directory Information: Files and Path 362
10.3 Creating and Deleting Files or Directories 373
10.4 Changing a File's Name or Other Attributes 376
10.5 About InputStreams/OutputStreams and Readers/Writers 379
10.6 Reading and Writing Files 381
10.7 Scanning Input with StreamTokenizer, Scanner, Parsers 384

10.8 Reading from the Standard Input or from the Console/Controlling Terminal 392
10.9 Copying a File 396
10.10 Reassigning the Standard Streams 397
10.11 Duplicating a Stream as It Is Written 398
10.12 Reading/Writing a Different Character Set 401
10.13 Those Pesky End-of-Line Characters 402
10.14 Beware Platform-Dependent File Code 403
10.15 Reading and Writing JAR or ZIP Archives 404
10.16 Reading Files in a Filesystem-Neutral Way with getResource() and getResourceAsStream() 409
10.17 Creating a Transient/Temporary File 411
10.18 Getting the Directory Roots 414
10.19 Using the File Watcher Service to Get Notified About File Changes 415
10.20 Walking a File Tree (like Find) 417

11. Threaded Java. 421
11.0 Introduction 421
11.1 Running Code in a Different Thread 423
11.2 Using Virtual Threads for Better Performance 429
11.3 Rendezvous and Timeouts 431
11.4 Synchronizing Threads with the synchronized Keyword 432
11.5 Simplifying Synchronization with Locks 436
11.6 Locking with One Writer, Many Readers 437
11.7 Sharing Data Among Threads—ThreadLocal and ScopedValue: Structuring Concurrency 440
11.8 Simplifying Producer/Consumer with the Queue Interface 444
11.9 Optimizing Parallel Processing with Fork/Join 447
11.10 Scheduling Tasks: Future Times, Background Saving in an Editor 451

12. Data Science and R. 455
12.0 Introduction 455
12.1 Using Data in Apache Spark 456
12.2 Using R Interactively 459
12.3 Comparing/Choosing an R Implementation 462
12.4 Using R from Within a Java App: Renjin 463
12.5 Using Java from Within an R Session 465
12.6 Using R in a Web App 467

13. Machine Learning/Artificial Intelligence. 469
13.0 Introduction 469
13.1 Some Major AI Software 472
13.2 Using ChatGPT Directly 476

13.3 Using ChatGPT via LangChain4j 479
13.4 Making an AI Service with LangChain4j 480
13.5 Conversing with Shadows 482
13.6 Generating Images with LangChain4j 484
13.7 Mixed Media Prompts: Inferences from Images with LangChain4j 486
13.8 Running AI Locally with ollama 488

14. Network Clients. 491
14.0 Introduction 491
14.1 HTTP/REST Web Client—Modern API **11** 494
14.2 Contacting a Socket Server 496
14.3 Finding and Reporting Network Addresses 497
14.4 Handling Network Errors 500
14.5 Reading and Writing Textual Data 501
14.6 Reading and Writing Binary or Serialized Data 504
14.7 Postcards of the Internet: Using UDP Datagrams 507
14.8 URI, URL, or URN? 510
14.9 Program: Sockets-Based Chat Client 511

15. Server-Side Java. 513
15.0 Introduction 513
15.1 Opening a Server Socket for Business 514
15.2 Finding Network Interfaces 517
15.3 Returning a Response (String or Binary) 518
15.4 Handling Multiple Clients 522
15.5 Serving the HTTP Protocol 526
15.6 Securing a Web Server with TLS (formerly SSL) and JSSE 529
15.7 Creating a REST Service/Microservice with JAX-RS 532
15.8 Unix Domain Sockets—Even on Windows! **16** 535

16. Processing JSON Data. 539
16.0 Introduction 539
16.1 Generating JSON Directly 541
16.2 Parsing and Writing JSON with Jackson 542
16.3 Parsing and Writing JSON with org.json 544
16.4 Parsing and Writing JSON with JSON-B 546
16.5 Finding JSON Elements with JSON Pointer 548

17. Reflection, or "A Class Named Class". 551
17.0 Introduction 551
17.1 Loading and Instantiating a Class Dynamically 552
17.2 Printing Class Information 554
17.3 Getting a Class Descriptor 556

17.4 Finding and Using Methods and Fields 557
17.5 Invoking Class Members via MethodHandles 562
17.6 Listing Classes in a Package 563
17.7 Accessing Nested Members of Same Class 565
17.8 Accessing Private Methods and Fields via Reflection 567
17.9 Constructing a Class from Scratch with a ClassLoader 568
17.10 Constructing a Class from Scratch with JavaCompiler 570
17.11 Constructing or Modifying Class Files with the Class-File API 22P 573
17.12 Using and Defining Annotations 577
17.13 Finding Plug-In-Like Classes via Annotations 581
17.14 A Timing Program 583
17.15 Program: CrossRef 585

18. Using Java with Other Languages. 587
18.0 Introduction 587
18.1 Running an External Program from Java 588
18.2 Running a Program and Capturing Its Output 592
18.3 Calling Other Languages via javax.script 596
18.4 Mixing Languages with GraalVM 21 599
18.5 Calling Between Java and Native Code with the Foreign Function and Memory API 22 601
18.6 Calling Other Languages via Native Code (JNI) 605
18.7 Calling Java from Native Code with JNI 611

Afterword. 615

A. Java Then and Now. 617

Index. 627

Preface

Like any of the most-used programming languages, Java has its share of detractors, advocates, issues, quirks,[1] and a learning curve. The *Java Cookbook* aims to help the Java developer get up to speed on some of the most important parts of Java development. I focus on the standard APIs and some third-party APIs, but I don't hesitate to cover language issues as well.

This is the fifth edition of this book, and it has been shaped by many people and by the myriad changes that Java has undergone over its two and a half decades of popularity. Readers interested in Java's history can refer to Appendix A.

Java 21 is the current long-term supported (LTS) version, and Java 24 is the latest current release at the time of publication of this edition. The current cadence of releases every six months may be great for the Java SE development team at Oracle, for developers who want access to the latest-and-greatest, and for click-driven, Java-related news sites, but it "may cause some extra work" for Java book authors, since books typically have a longer revision cycle than Java now does! Java 9, which came out after a previous edition of this book, was a sort-of-breaking release, the first release in a very long time to almost break backward compatibility, primarily the Java Platform Module System.

Everything in this edition of the book can be assumed to work on Java 11 or later, unless specified otherwise (see "Icons" on page xxi). At this point in time, nobody should be using Java 8 (or anything before it) for anything. Certainly nobody should be doing new development in Java 8. If you are, it's time to move on! While Java 11 is considered an LTS release, it is so old that you shouldn't use it. Java 17 is the previous LTS and quite usable, but if you want the best start on an LTS Java release, begin with Java 21.

1 For the quirks, see the *Java Puzzlers* books by Joshua Bloch and Neal Gafter (Addison-Wesley).

The goal of this revision of *Java Cookbook* is to keep the book up to date with all this change. I've included numerous new features, which meant I had to remove a significant amount of material. There are numerous additions throughout the chapters. In several places, a few recipes have been combined into one in the interest of simplicity and readability (and reducing page count). And of course I've updated a lot of other information along the way.

Who This Book Is For

I'm going to assume that you know the basics of Java. I won't tell you how to `println` a string, nor how to write a class that extends another and/or implements an interface. I'll assume you've taken a Java course such as Learning Tree's Introduction (which I wrote the first version of, decades ago) or worked your way through an introductory book such as:

- *Head First Java* by Kathy Sierra and Bert Bates (O'Reilly), a tutorial filled with visual metaphors and other brain-friendly learning techniques
- *Java in a Nutshell* by David Flanagan (O'Reilly), a more concise book covering the standard APIs in more detail
- *Think Java* by Allen Downey and Chris Mayfield (O'Reilly), a hands-on introduction to computer science and programming via Java
- *Learning Java* by Marc Loy, Patrick Niemeyer, and Daniel Leuck (O'Reilly), a book for practicing programmers moving to Java

However, Chapter 1 covers some techniques that you might not know very well and that are necessary to understand some of the later material. Feel free to skip around! Both the printed version of the book and the electronic copy are heavily cross-referenced.

Just to give an idea of how big a topic Java really is, have a look at Figure P-1. An earlier version of this diagram was online at Sun, prior to the Oracle takeover, with each box being a link to the corresponding documentation. Figure P-1 is my recreation of it, using a simple Java Swing program written for the occasion.

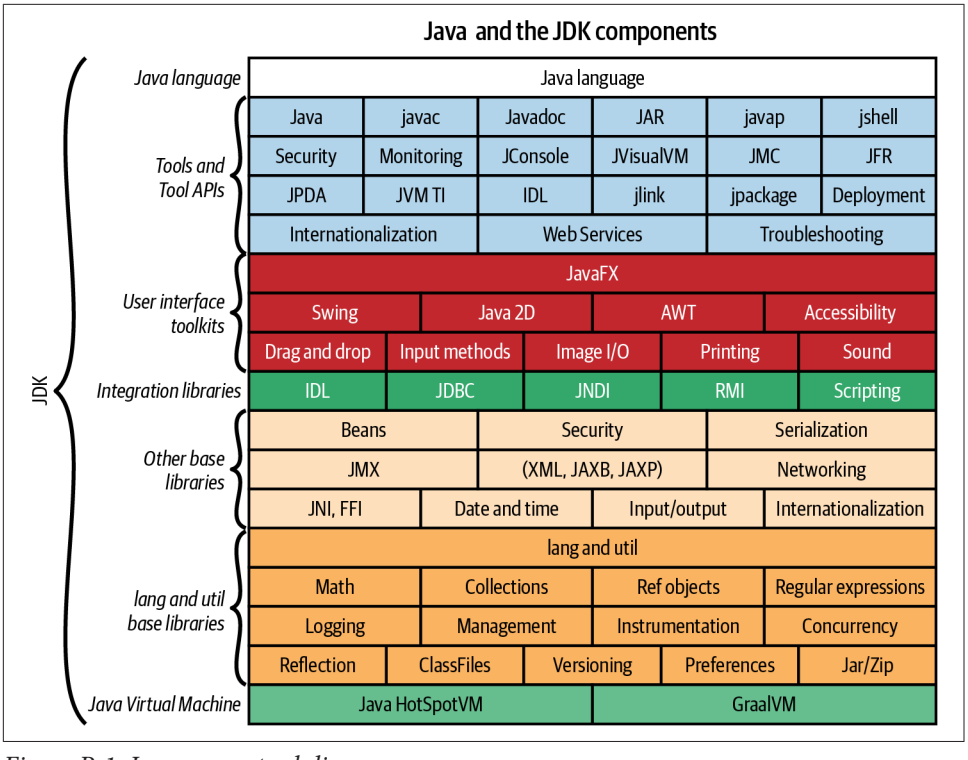


Figure P-1. Java conceptual diagram

What's in This Book?

In the early 2000s, Java seemed better suited to "development in the large," or enterprise application development, than to the few-line, one-off script in Perl, AWK, Python, or the Unix/Linux shells. That's because it is a compiled, object-oriented language. However, Java has gotten better for ad hoc tasks that might previously have been done with a scripting language. JShell (Recipe 1.5) and the run-java-source-file mechanism (see "Discussion" on page 2) have helped move Java in this direction. I illustrate many techniques with shorter Java class examples and even code fragments; some of the simpler ones will be shown using JShell. All of the code examples over a few lines long are in one of my public GitHub repositories, so you can rest assured that all the code you see here has been compiled and run (most of them run recently).

What's Not in This Book

Since some readers will have used previous editions (my thanks to you!), here's a brief list of information that is no longer included.

The fifth edition removed obsolete recipes from several chapters. The longer program listings at the end of many chapters were removed from the book, replaced by links to view them either from the code repository or online.

The fourth edition removed the discussion of Swing GUI and AWT Graphics, though these are still used in some examples. The code for database access was also removed. The code for the former lives on in the *desktop* project within the *javasrc* repository on GitHub. For the latter, see my *jpademo* GitHub repo.

The removed recipes have been migrated to my website.

Organization of This Book

Let's go over the organization of this book. Each chapter consists of a handful of recipes—short sections that describe a problem and its solution, along with a code example. The code in each recipe is intended to be largely self-contained; feel free to borrow bits and pieces of any of it for use in your own projects. The code is distributed with a Berkeley-style "use with attribution" copyright. Most recipes provide reasonably complete coverage of their topic. Some topics, however, have more detail, more nuance, than can be covered in the recipe format; these end with references that the reader can consult for additional explanation and examples.

I start off Chapter 1, "Getting Started: Compiling and Running Java", by describing some methods of compiling your program on different platforms and running them in different environments (command line, IDE, and windowed desktop). This is followed by Chapter 2, "Software Development, Testing, and Maintenance", which discusses techniques for building real applications. Topics such as Java modules, build tools, unit testing, and continuous integration are discussed here.

The next few chapters deal with basic APIs. Chapter 3, "Strings and Things", concentrates on one of the most basic but powerful data types in Java, showing you how to assemble, dissect, compare, and rearrange "strings," what you might otherwise think of as ordinary text. This chapter also covers the topic of internationalization/localization so that your programs can work as well in Akbar, Afghanistan, Algiers, Amsterdam, and Angleterre as they do in Alberta, Arkansas, and Alabama.

Chapter 4, "String Matching with Regular Expressions", teaches you how to use the powerful regular expressions technology from Unix on strings, in many string-matching and pattern-matching problem domains. Regex processing has been standard in Java for years, but if you don't know how to use it, you may be reinventing the flat tire.

Chapter 5, "Numbers", deals both with built-in numeric types such as `int` and `double`, as well as the corresponding API classes (`Integer`, `Double`, etc.) and the conversion and testing facilities they offer. There is also brief mention of the "big number" classes.

Because Java programmers often need to deal in dates and times, both locally and internationally, Chapter 6, "Dates and Times", covers this important topic.

The next chapter covers structuring your program's data. As in most languages, arrays in Java are linear, indexed collections of similar objects, as discussed in Chapter 7, "Structuring Data with Java". This chapter goes on to deal with the many Collections classes: powerful ways of storing quantities of objects in the java.util package, including the use of Java Generics.

Despite some syntactic familial resemblance to procedural languages such as C, Java is at heart an object-oriented programming (OOP) language, with some important functional programming (FP) constructs skilfully blended in. Chapter 8, "Object-Oriented Techniques", discusses some of the key notions of OOP as it applies to Java, including the commonly overridden methods of `java.lang.Object` and the important issue of design patterns. Java is not, and never will be, a pure FP language. However, it is possible to use some aspects of FP, increasingly so with Java 8 and its support of lambda expressions (a.k.a. closures). This is discussed in Chapter 9, "Functional Programming Techniques: Functional Interfaces, Streams, and Parallel Collections".

Chapter 10, "Input and Output: Reading, Writing, and Directory Tricks", details the rules for reading and writing files (don't skip this if you think files are boring; you'll need some of this information in later chapters). The chapter also shows you everything else about files—such as finding their size and last-modified time—and about reading and modifying directories, creating temporary files, and renaming files on disk.

Java was one of the first mainstream languages to provide language support for *threads*, the notion of multiple flows of control within a single process. Chapter 11, "Threaded Java", tells you how to write classes that appear to do more than one thing at a time and let you take advantage of powerful multiprocessor hardware.

Big data and data science have become a thing, and Java is right in there. Chapter 12, "Data Science and R" covers some key data science software and the R language. Apache Hadoop, Apache Spark, and much more of the big data infrastructure is written in, and extensible with, Java. Spark is also introduced in this chapter. The R programming language is popular with data scientists, statisticians, and other scientists. R, like Bell Labs' original S language it was copied from, was implemented in C/C++. At least two reimplementations of R have been written in Java, and Java can also be interfaced directly with the standard R implementation in both directions.

Machine learning (ML) along with generative large language models (LLMs) are key parts of what is termed artificial intelligence (AI). AI has become a hot topic since the release of ChatGPT and its competitors and offspring. Chapter 13, "Machine Learning/Artificial Intelligence", talks about AI in general and then focuses on several Java libraries for applying AI.

Because Java was originally promulgated as the programming language for the internet, it's only fair to spend some time on networking in Java. Chapter 14, "Network Clients", covers the basics of network programming from the client side, focusing on sockets. Today, so many applications need to access a web service, primarily RESTful web services, that this seemed necessary. I'll then move to the server side in Chapter 15, "Server-Side Java", wherein you'll learn some server-side programming techniques.

One simple text-based representation for data interchange is JSON, the JavaScript Object Notation. Chapter 16, "Processing JSON Data", describes the format and some of the many APIs that have emerged to deal with it.

Chapter 17, "Reflection, or "A Class Named Class"", lets you in on such secrets as how to write API cross-reference documents mechanically and how web servers are able to load any old servlet—even if they've never seen that particular class before—and run it.

Sometimes you already have code written and working in another language that can do part of your work for you, or you want to use Java as part of a larger package. Chapter 18, "Using Java with Other Languages", shows you how to run an external program (compiled or script) and also interact directly with native code in C/C++ or other languages.

There isn't room in a book this size for everything I'd like to tell you about Java. The brief Afterword presents some closing thoughts and a link to my online summary of Java APIs that every Java developer should know about.

Finally, Appendix A, "Java Then and Now", gives the storied history of recent Java in a release-by-release timeline, so whatever version of Java you learned, you can jump in here and get up to date quickly. Versions prior to 16 have been relocated to my own website.

So many topics, and so few pages! Many topics do not receive 100% coverage; I've tried to include the most important or most useful parts of each API. To go beyond, check the official *Javadoc* pages for each package; many of these pages have some brief tutorial information on how the package is to be used.

Besides the parts of Java covered in this book, two other platform editions, Java Micro Edition (Java ME) and Java Enterprise Edition (Java EE), have been standardized. Java ME is concerned with small devices such as handhelds, tiny "feature phones" (not really smartphones), fax machines, and the like. At the other end of the size scale—large server machines—there's Eclipse Jakarta EE. This is the follow-on to the former Java EE, which in the last century was briefly known as J2EE. Jakarta EE is concerned with building large, scalable, distributed applications. APIs that are part of Jakarta EE include Servlets, JavaServer Pages, JavaServer Faces, JavaMail, Contexts and Dependency Injection (CDI), JAX-RS for RESTful Web Services, JSON-P and JSON-B for processing JSON-format data, and Transactions. Jakarta EE package names originally began with "javax." because they are not core packages, but have now moved to "jakarta." This book mentions only a few of these. *Jakarta EE Cookbook* by Elder Moraes (Packt Publishing) covers some of the Jakarta EE APIs, as does the older *Java Servlet and JSP Cookbook* by Bruce Perry (O'Reilly).

This book doesn't cover Java ME at all. But speaking of cell phones and mobile devices, you probably know that Android uses Java as its language, as well as Kotlin, which is a simplification and extension of Java. What should be comforting to Java developers is that Android also uses most of the core Java API, except for Swing and AWT, for which it provides Android-specific replacements. The Java developer who wants to learn Android may consider looking at my *Android Cookbook* (O'Reilly), or the book's website.

Java Books

A lot of useful information is packed into this book. However, due to the breadth of topics, it is not possible to give book-length treatment to any one topic. Because of this, the book contains references to many websites and other books. In pointing out these references, I'm hoping to serve my target audience: the person who wants to learn more about Java.

O'Reilly publishes, in my opinion, the best selection of Java books on the market. As the API continues to expand, so does the coverage. Check out the complete list of O'Reilly's collection of Java books; you can buy them at most bookstores, both physical and virtual. You can also read them online through the O'Reilly Online Learning Platform, a paid subscription service. Though many books are mentioned at appropriate spots in this book, a few deserve special mention. However, to keep this book short, and allow updating, I've created a couple of book lists, one Java specific and one for developers in general (not to mention one for "the educated adult").

Conventions Used in This Book

This book uses the following conventions.

Programming Conventions

I use the following terminology in this book. A *program* means any unit of code that can be run, from a three-line main program to an enterprise component or a full-blown GUI application. An *application* is another name for a program. A *desktop application* (a.k.a. client) interacts with the user. A *server program* deals with a client indirectly, usually via a network connection (usually REST over HTTPS these days).

The examples shown are in two varieties. Those that begin with zero or more import statements, a Javadoc comment, and often a `public class` statement are complete examples. Those that begin with a declaration or executable statement, of course, are excerpts. However, the full versions of all code samples have been compiled and run, and the online source includes the full versions.

Recipes are numbered by chapter and number, so, for example, Recipe 8.1 refers to the first recipe in Chapter 8.

Typesetting Conventions

The following typographic conventions are used in this book:

Italic
: Used for commands, filenames, and example URLs. It is also used for emphasis and to define new terms when they first appear in the text.

`Constant width`
: Used in code examples to show partial or complete Java source code program listings. It is also used for class names, method names, variable names, and other fragments of Java code.

`Constant width bold`
: Used for user input, such as commands that you type on the command line.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Icons

This element signifies a tip or suggestion.

This element signifies a general note.

This element signifies a warning or caution.

Sidebars

Paragraphs like this with vertical bars on each side indicate a "sidebar," a discussion that isn't part of any one recipe but may apply to several recipes in the same part of the text.

25 This icon (with a particular release number) indicates the minimum Java version for a given explanation or code sample. A *P* after the number indicates that it is a preview feature, thus `--enable-preview` is needed. Similarly, a *C* after the number indicates an incubating feature, requiring `--add-modules jdk.incubator.MMM` (where `MMM` is the incubation module, e.g., `concurrent`, `vector`, etc.).

Incubation, Preview, and Production

The Oracle team behind Java are very deep thinkers when it comes to language features, and will never rush a feature into the language, nor let it in without adequate consideration. In order to enable careful phase-in of new functionality, Java puts significant new features into one of two "trial" or "upcoming" states: *preview* and *incubation*. Preview features are nearly final and usually make it into Java after a few releases, whereas incubator features are less complete and more tentative. Brian Goetz explains the difference between preview and incubation:

> Preview features are really finished but are waiting for a round of feedback, whereas the incubator mechanism has more room to iterate over the API several times to get feedback.

However, the "string template" preview feature from Java 21 and 22 was withdrawn for rethinking in Java 23, so it presumably wasn't "really finished." Preview features must be enabled with the `--enable-preview` feature, both during compile and execution, and also `--source 21` (or other version number) when compiling (but not when running compiled code). Incubator modules require `--incubation` *`MODULENAME`* and usually a *module-info* file that depends on the given module.

Another key difference is that preview features provide imports from their expected final package names, so you likely won't have to change your code when the feature comes out of preview. Incubator features, on the other hand, always provide imports from `jdk.incubator.`*`MODULENAME`*, meaning you definitely will need to change your code (at least the imports) when the feature progresses to preview or to production in a future release.

Using Code Examples

The code samples for this book are in my GitHub repositories. Most are in the *javasrc* repo, but a few are pulled in from another repository, *darwinsys-api*. Details on downloading these are in Recipe 1.3.

Many programs are accompanied by an example showing them in action, run from the command line. These will usually show a prompt ending in either `$` for Unix/Linux/macOS, or `>` for Windows, depending on which computer I happened to be using the day I wrote that example. Text before this prompt character can be ignored; it may be a pathname or a hostname, again, depending on the system.

These examples will usually also show the full package name of the class because Java requires this when starting a compiled program from the command line. Because that will remind you which subdirectory of the source repository contains the source code, I won't point it out explicitly very often.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Java Cookbook* by Ian F. Darwin (O'Reilly). Copyright 2025 RejmiNet Group, Inc., 978-1-098-16997-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *https://oreilly.com*.

Comments and Questions

As mentioned earlier, I have tested all the code on at least one of the reference platforms, and most on several. Still, there may be platform dependencies, or even bugs, in my code or in some important Java implementation. Please report any errors you find, as well as your suggestions for future editions. We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/java-cookbook-5ed*. You can also email *bookquestions@oreilly.com* to comment or ask technical questions about this book. If all else fails, you can contact O'Reilly:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

For more information about our books, courses, conferences, and news, see our website at *https://www.oreilly.com*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

The O'Reilly site lists errata. You'll also find the source code for all the Java code examples to download; *please* don't waste your time typing them again! For specific instructions, see Recipe 1.3.

Acknowledgments

I wrote in the Afterword to the first edition that "writing this book has been a humbling experience." I should add that maintaining it has been humbling, too. While many have been lavish with their praise—one very kind reviewer called it "arguably the best book ever written on the Java programming language"—I have been humbled by the number of errors and omissions in earlier editions. I have endeavored to correct these.

My life has been touched many times by the flow of the fates bringing me into contact with the right person to show me the right thing at the right time. Steve Munro, who is now an oft-cited Toronto Transportation critic, introduced me to computers when we were in the same class in high school—in particular an IBM 360/30 at the Toronto Board of Education that was bigger than a living room, had 32 or 64*K* (not M or G!) of memory, and had perhaps less computing power than a smartwatch. The late Herb Kugel took me under his wing at the University of Toronto while I was learning about the larger IBM mainframes that came later. Terry Wood and Dennis Smith at the University of Toronto introduced me to mini- and micro-computers before there was an IBM PC. On evenings and weekends, the Toronto Business Club of Toastmasters International and Al Lambert's Canada SCUBA School allowed me to develop my public speaking and teaching abilities. Several people at the University of Toronto, but especially Geoffrey Collyer, taught me the features and benefits of the Unix operating system at a time when I was ready to learn it.

Thanks to the many Learning Tree instructors and students who showed me ways of improving my presentations. I still teach for "The Tree" and recommend their courses for the busy developer who wants to zero in on one topic in detail over a few days.

Closer to this project, Tim O'Reilly believed in my "little Lint book" when it was just a sample chapter from a proposed longer work, enabling my early entry into the rarefied circle of O'Reilly authors. Years later, Mike Loukides encouraged me to keep trying to find a Java book idea that both he and I could work with. And he stuck by me when I kept falling behind the deadlines. Mike also read the entire manuscript and made many sensible comments, some of which brought my flights of fancy down to earth. Jessamyn Read turned many scratchings of dubious legibility into the quality illustrations you see in this book. And many, many other talented people at O'Reilly helped put this book into the form in which you now see it.

The code examples are dynamically included (so updates get done faster) rather than pasted in. My son (and functional programming developer) Benjamin Darwin helped meet the deadline for an earlier edition by converting almost the entire codebase to O'Reilly's newest "include" mechanism and by resolving a couple of other non-Java presentation issues. He also helped make Chapter 9 clearer and more functional.

At O'Reilly

For this fifth edition of the book, my sincere thanks to all the talented people who worked on the manuscript; they're all listed on the publication data page. For the previous (fourth) edition of the book, Suzanne McQuade was the editorial overseer, and Corbin Collins the principal editor. Corbin was especially meticulous in checking the manuscript. Meghan Blanchette, Sarah Schneider, Adam Witwer, Melanie Yarbrough, and the many production people who played a part in getting the third edition ready for you to read. Thanks to Mike Loukides, Deb Cameron, and Marlowe Shaeffer for editorial and production work on the second edition.

Technical Reviewers

For the fifth edition, I received a great deal of quality feedback from George Ball, Jason Clark, Marc Loy, Nighi Nair, and Andrew Slice. George, Jason, and Marc in particular sent lots of specific comments on the manuscript and tried to browbeat me into not letting anything vague make it into the book. Thank you all for the time you took in preparing your feedback. I'm sorry I didn't get 100% of your suggestions in, but each of you has improved the book in many ways, which you may recognize. The errors that remain are on me.

For the fourth edition I was blessed to have two very thorough technical reviewers, Sander Mak and Daniel Hinojosa. Many issues that I hadn't considered during the main revision were called out by these two, leading to extensive rewrites and changes

in the last few weeks before the O'Reilly production team took over. Thanks so much to both of you!

My reviewer for the third edition, Alex Stangl, read the third edition manuscript and went far above the call of duty, making innumerable helpful suggestions, even finding typos that had been present in previous editions! Helpful suggestions on particular sections were made by Benjamin Darwin, Mark Finkov, and Igor Savin. For anyone I've forgotten to mention, I thank you all!

Bil Lewis and Mike Slinn made helpful comments on multiple drafts of the first edition. Ron Hitchens and Marc Loy carefully read the entire final draft of the first edition. I am grateful to Mike Loukides for his encouragement and support throughout the process. Editor Sue Miller helped shepherd the manuscript through the somewhat energetic final phases of production. Sarah Slocombe read the XML chapter in its entirety and made many lucid suggestions; unfortunately, time did not permit me to include all of them in the first edition.

Jonathan Knudsen, Andy Oram, and David Flanagan commented on the book's outline when it was little more than a list of chapters and recipes, and they were able to see the kind of book it could become and suggest ways to make it better.

Each of these people made this book better in many ways, particularly by suggesting additional recipes or revising existing ones. Thanks to one and all! The faults that remain are my own.

Readers

My sincere thanks to all the readers who found errata and suggested improvements. Every new edition is better for the efforts of folks like you who take the time and trouble to report that which needs reporting!

Special mention must be made of one of the book's German translators,[2] Gisbert Selke, who read the first edition cover to cover during its translation and clarified my English. Gisbert did it all over again for the second edition and provided many code refactorings, which made this a better book than it would be otherwise. Going beyond the call of duty, Gisbert even contributed one recipe that appeared in several previous editions, and revised some of the other recipes in the same chapter. Thank you, Gisbert!

The second edition also benefited from comments by Jim Burgess, who read large parts of the book. Comments on individual chapters were received from Jonathan

2 Earlier editions are or have been available in English, German, French, Polish, Russian, Korean, Traditional Chinese, and Simplified Chinese. My thanks to all the translators for their efforts in making the book available to a wider audience.

Fuerth, the late Kim Fowler, Marc Loy, and Mike McCloskey. My wife, Betty, and my then-teenaged children each proofread several chapters as well.

The following people contributed significant bug reports or suggested improvements: Rex Bosma, Rod Buchanan, John Chamberlain, Keith Goldman, Gilles-Philippe Gregoire, B. S. Hughes, Jeff Johnston, Rob Konigsberg, Tom Murtagh, Jonathan O'Connor, Mark Petrovic, Steve Reisman, Bruce X. Smith, and Patrick Wohlwend.

Etc.

My dear wife, Betty Cerar, still knows more about the caffeinated beverage that I drink while programming than the programming language I use, but her passion for clear expression and correct grammar has benefited so much of my writing during our life together.

No book on Java would be complete without a note of thanks to James Gosling for inventing Java (he also invented the first Unix Emacs, the *sc* spreadsheet, and the NeWS window system). Thanks also to his employer Sun Microsystems (before they were taken over by Oracle) for releasing not only the Java language but also an incredible array of Java tools and API libraries freely available over the internet.

Willi Powell of Apple Canada provided macOS access for the first edition, back in the early days of macOS.

To each and every one of you, my sincere thanks.

Book Production Software

I used a variety of tools and operating systems in preparing, compiling, and testing this book. The developers of OpenBSD, "the proactively secure Unix-like system," deserve thanks for making a stable and secure Unix clone that is also closer to traditional Unix than other freeware systems. I used the *vi* editor (*vi* on OpenBSD and *vim* on Windows) while inputting the original manuscript in XML, and I used Adobe FrameMaker to format the documents. Frame was a wonderful GUI-based documentation tool that Adobe bought and subsequently "destroyed"; they promised to make it available for macOS but failed to do so for years after macOS 10.0.

The third, fourth, and fifth editions of this *Java Cookbook* were formatted in Asciidoctor and brought to life on the publishing toolchain of O'Reilly's Atlas.

CHAPTER 1

Getting Started: Compiling and Running Java

1.0 Introduction

This chapter covers some entry-level tasks that you need to know how to do before you can go on. It is said you must crawl before you can walk, and walk before you can ride a bicycle. Before you can try out anything in this book, you need to be able to compile and run your Java code. I start by demonstrating several methods: using the Java Development Kit (JDK), an Integrated Development Environment (IDE), and build tools like Maven and Gradle. Another issue people run into is setting `CLASSPATH` correctly, so that's dealt with next. Deprecation warnings follow after that, because you're likely to encounter them in maintaining old Java code. The chapter ends with some general information about unit testing and assertions.

If you don't have Java installed, you'll need either to download the JDK on its own, particularly to the command-line Java, or download an IDE that includes its own JDK. Be aware that in historical releases there were different downloads, the Java Runtime Environment (JRE) and the JDK. Up until Java 8, the JRE was a smaller download for end users (i.e., just the parts needed to run a Java app, but not to compile one). Since there is less desktop Java than there once was, and since hard disks have gotten much bigger, the JRE was eliminated in favor of creating `jlink` as part of the JDK, to create custom downloads of the JDK (see Recipe 2.17). The JDK download (sometimes referred to as the Java Software Development Kit or SDK) is the full development environment, which you'll want if you're going to develop Java software.

Standard downloads for the current release of Java are available at Oracle's website and from several other sites (not a complete list):

- Eclipse Software Foundation
- Microsoft
- Azul
- Java.net

The entire JDK is maintained as an open source project, and the OpenJDK source tree is used (with changes and additions) to build the commercial and supported Oracle JDKs. The OpenJDK repository has the source code for the entire JDK. You can in fact build your own JDK fairly easily from this GitHub repository; see my article on the JDK source code, about halfway down the page.

If you are going to have more than one version of the JDK on your Unix/Linux/macOS computer, you might want to investigate SDKMAN, which lets you easily install and switch between them.

If you're already happy with your IDE, you may wish to skip some or all of the next few recipes. The coverage is here to ensure that everybody can compile and debug their programs before we move on.

1.1 Hello, World: Compiling and Running Java with the Standard JDK

Problem

You need to compile and run your Java program.

Solution

This is one of the few areas where your computer's operating system impinges on Java's portability, so let's get these issues out of the way first.

Discussion

Using the command-line JDK may be the best way to keep up with the very latest improvements in Java. Assuming you have the standard JDK installed in the standard location and/or have set its location in your `PATH`, you should be able to run the command-line JDK tools. Use the commands `javac` to compile and `java` to run your program (and, on Windows only, `javaw` to run a program without a console window), like this:

```
C:\> javac HelloWorld.java

C:\> java HelloWorld
Hello, World
```

```
C:\>
```

If the program refers to other classes for which the source is available (in the same directory) and a compiled *.class* file is not, `javac` will automatically compile it for you. Effective with Java 11, for simple programs that don't need any such co-compilation, you can combine the two operations—compiling and executing—simply by passing the Java source file to the `java` command:

```
$ java HelloWorld.java
Hello, Java
$
```

22 Effective with Java 22, the compile-and-run form of the `java` command will find and compile additional source files. Try this on Java 22+:

```
$ cd main/src/main/java/starting
$ ls ???.java
One.java  Two.java
$ java One.java
```

Don't be confused by the compiler's (lack of) output in these examples. Both `javac` and `java` work on the Unix "no news is good news" philosophy: if a program was able to do what you asked it to, it shouldn't bother chattering at you to say that it did so. In the words of W.H. Auden, "had anything been wrong, we should certainly have heard."

There is an optional setting called `CLASSPATH`, discussed in Recipe 1.6, that controls where Java looks for classes. `CLASSPATH`, if set, is used by both `javac` and `java`. In older versions of Java, you had to set your `CLASSPATH` to include "." even to run a simple program from the current directory; this is no longer true on current Java.

Sun/Oracle's `javac` compiler is the official reference implementation. There were historically several alternative open source command-line compilers, including Jikes and Kaffe, but they are, for the most part, no longer actively maintained.

There have also been some Java runtime clones, including Apache Harmony, Japhar, the IBM Jikes Runtime, and even JNode, a complete, standalone operating system written in Java. Since the Sun/Oracle JVM has been released as open source software as OpenJDK under the GNU Public License (GPL), most of these projects are no longer maintained. Harmony, for example, was retired by Apache in 2011. Today, a dozen or so organizations supply builds of OpenJDK for free download (and use). Some are enhanced, including Oracle's commercial version and Azul Platform Prime, a heavily optimized derivative of OpenJDK. One alternative implementation that is still maintained is IBM's J9, (a reimplementation, descended from a runtime that existed before Java, and now hosted at *Eclipse.org*).

1.2 Hello, World of Classless Main 21P

Problem

It is tedious to type out `public class Foo {` and `public static void main (String[] args)` every time you start a new program.

Solution

Use the preview implicitly declared class and instance main features.

Discussion

Implicitly declared class refers to a main program declared by itself, with no enclosing *class* structure. *Instance main* refers to a nonstatic `main()` method (with or without a `String[]` argument); such methods are runnable from the command line or an up-to-date IDE, just like traditional `main()` methods. For convenience, I'll lump these together under the unofficial term *classless main*.

A simple main program can now be constructed with as little as:

```
void main() {
        System.out.println("Hello, world");
}
```

It may strike joy or fear into the hearts of long-time Java developers and educators (and book authors), but the preceding is a complete compilation unit. Assume it's stored in a file called *hello.java*. Due to the joys of the source-code launcher (another recent Java feature), we can compile and run this program using just:

```
$ java hello.java
```

At least, we'll be able to when the implicit class and instance main features come out of preview. For now we must type:

```
$ java --enable-preview --source 22 hello.java
```

That's almost more typing than the original form of the Hello program! Fortunately, if you're doing this command-line style on Unix, Linux, macOS, or on Windows with the `git-bash` add-on, you can save this as an alias in your shell startup file. I named it `javapvr` for "java (with preview) run," as this alias cannot be used to run compiled *.class* files; the `--source` is interpreted as meaning that you want to compile and run a *.java* file:

```
alias javapvr='java --enable-preview --source 22'
```

If running in an IDE or build tool, the same options must be applied. In build tools, apply both options to the compilation (but only the `--enable-preview` to the run

step). IntelliJ IDEA, for example, when it detects a preview feature, just asks you if it's OK to enable Preview for the given IDE project.

Then you can "live in the future" and just use the `javapvr` command to run *.java* files. And when the future arrives—these two features come out of preview—you can drop the alias and just use the plain `java` command.

Note that any compile or run of a program using preview features will generate a few lines of warnings to remind you that it is a preview. I have removed these warnings from many of the example outputs, to focus on the code and its output.

This implicitly declared class and instance main mechanism is intended as an on-ramp to those getting started with Java. As such, it has a few other differences from "the way we were":

- An implicit class cannot have a package declaration, but may of course have imports.
- An implicit class does not need to have a capitalized filename(!); the generated class name will be taken from the filename (see Example 1-1).
- An instance main does not need to declare its `String[]` argument, but may if it needs access to the command-line arguments.
- An instance main is not static and can call nonstatic methods.

All of this is designed to make it easier to get started with Java; educators no longer need to handwave or explain away all the traditional boilerplate code before beginners can get started writing Java code. *This is huge!*

The longer version shown in Example 1-1 shows more of the pieces of classless main. Again, this is the complete compilation unit.

Example 1-1. main/src/main/java/starting/HelloClasslessWithMembers.java

```
import java.lang.reflect.*;  // Unused, just to show import is allowed

int x;            // instance fields allowed

void main(String[] args) {  // instance main
  System.out.println("Hello, world");
  process(1);
}
void process(int n) {    // instance method
  System.out.println("Hello " + n);
}
```

This prints out a couple of Hello messages:

```
C:\> java --enable-preview --source 21 HelloClasslessWithMembers.java
Hello, world
Hello 1
```

We can still compile this like any other Java program. As it's still in preview, we need to add the two command-line arguments used earlier into the compile command. The generated class file can be examined with the standard JDK tool `javap`, and run with the standard `java` command (with the `--enable-preview` argument):

```
$ javac --enable-preview --source 21 HelloClasslessWithMembers.java
$ javap HelloClasslessWithMembers
Compiled from "HelloClasslessWithMembers.java"
final class HelloClasslessWithMembers {
  int x;
  HelloClasslessWithMembers();
  void main(java.lang.String[]);
  void process(int);
}
$ java --enable-preview HelloClasslessWithMembers
Hello, world
Hello 1
```

Note that neither `main()` nor `process()` is static; this is a feature of instance main. These features are described in JEP 445: "Unnamed Classes and Instance Main Methods (Preview)".[1] These two features are both in preview as of Java 22; I hope they will become mainstream as quickly as possible, as it will make it so much easier to teach Java in the future!

23P In Java 23, the updated version of this preview feature allows even more simplification, the omission of `System.out` in calls to the ubiquitous `println()`:

```
void main() {
        println("Hello world");*
}
```

The elimination of the need to explain `System.out` to Java beginners is another forward step along the road to making Java easier to learn. It's a step that I and others had petitioned the Java team to include (after initially rejecting the idea, they came up with a better, cleaner way of implementing it). The easier-to-use `println()` works by having the compiler auto-import statically the methods from the new class `java.io.IO`, which simply delegates `println()` and `print()` to `System.out`, and `readline()` to the `Console` class described in Recipe 10.8. Again, this is a preview feature and therefore subject to change.

1 A JEP is a Java Enhancement Proposal, part of the OpenJDK project's processes for "moving Java forward." There's a discussion at *https://www.infoworld.com/article/2335544*.

Further, the Java 23 preview provides on-demand auto-importing of everything from the `java.base` module, that is, much of the standard API.

See Also

Implicitly declared classes, instance main, and `java.io.IO` are discussed with their rationale in the JEP 477 for Java 23 (but only for implicitly declared classes)..

1.3 Downloading and Using the Code Examples

Problem

You want to try out my example code and/or use my utility classes.

Solution

Download the latest archive of the book source files, unpack it, and run Maven (see Recipe 2.4) to compile the files.

Discussion

The source code used as examples in this book is included in a couple of source code repositories that have been in continuous development since 1995. These are listed in Table 1-1.

Table 1-1. The main source code repositories

Repository name	GitHub URL	Package description	Approx. size
javasrc	*https://github.com/IanDarwin/javasrc*	Java code examples/demos	1,470 classes
darwinsys-api	*https://github.com/IanDarwin/darwinsys-api*	A published API	200 classes

You can download these repositories from the GitHub URLs shown in Table 1-1. GitHub allows you to download a ZIP file of the entire repository's current state, as well as view individual files on the web interface. Downloading with `git clone` instead of as an archive is preferred because you can then obtain all my updates at any time with a simple `git pull` command. And with the amount of updating this codebase has undergone for the current release of Java, you are sure to find changes after the book is published.

If you are not familiar with Git, see "CVS, Subversion, Git, Oh My!" on page 11.

javasrc

This is the largest repo and consists primarily of code written to show a particular feature or API. The files are organized into subdirectories by topic, many of which correspond more or less to book chapters—for example, a directory for *strings* examples (Chapter 3), *regex* for regular expressions (Chapter 4), *numbers* (Chapter 5), and so on. The archive also contains the index by name and index by chapter files from the download site, so you can easily find the files you need.

The javasrc library is further broken down into a dozen Maven modules (shown in Table 1-2) so that you don't need all the dependencies for everything on your `CLASSPATH` all the time.

Table 1-2. javasrc Maven modules

Directory/module name	Description	Reference
pom.xml	Maven parent POM	Recipe 2.4
desktop	AWT and Swing stuff (no longer covered in *Java Cookbook*)	*https://darwinsys.com/java/cookbookcuttings*
ee	Enterprise stuff (no longer covered in *Java Cookbook*)	*https://darwinsys.com/java/cookbookcuttings*
graal	GraalVM demos	Recipe 1.12
jlink	`jlink` demos	Recipe 2.17
json	JSON processing	Chapter 16
main	Contains the majority of the files, i.e., those not required to be in one of the other modules due to `CLASSPATH` or other issues	*passim*
Rdemo-web	R demo using a web framework	Recipe 12.2
restdemo	REST service demo	Recipe 14.1
spark	Apache Spark demo	Recipe 12.1
testing	Code for testing	Recipe 2.9
unsafe	Demo of `Unsafe` class	"Probably?" on page 53
xml	XML processing (no longer covered in *Java Cookbook*)	*https://darwinsys.com/java/cookbookcuttings*

darwinsys-api

I have built up a collection of useful stuff partly by moving some reusable classes from *javasrc* into my own API, which I use in my Java projects. I use example code from it in this book, and I import classes from it into numerous examples. If you want to use these classes in your own project, the easiest way is to add it to your *pom.xml* (or the corresponding information in `build.gradle`):

```
<dependency>
   <groupId>com.darwinsys</groupId>
```

```
    <artifactId>darwinsys-api</artifactId>
    <version>1.8.0</version>
</dependency>
```

The version number will change over time. You can search on Maven Central to see the latest, and to see the specification in the format for other build tools.

This API consists of about two dozen com.darwinsys packages, listed in Table 1-3. The structure vaguely parallels the standard Java API; this is intentional. These packages now include around 200 classes and interfaces. Most of them have Javadoc documentation that can be viewed with the source download.

Table 1-3. The com.darwinsys packages

Package name	Package description
com.darwinsys.calendar	Calendar/scheduling
com.darwinsys.csv	Classes for comma-separated values files
com.darwinsys.database	Classes for dealing with databases in a general way
com.darwinsys.diff	Comparison utilities
com.darwinsys.formatting	Various formatting helpers
com.darwinsys.genericui	Generic GUI stuff
com.darwinsys.geo	Classes relating to country codes, provinces/states, and so on
com.darwinsys.graphics	Graphics
com.darwinsys.html	Classes (only one so far) for dealing with HTML
com.darwinsys.io	Classes for input and output operations, using Java's underlying I/O classes
com.darwinsys.jclipboard	The `JClipBoard` class
com.darwinsys.lang	Classes for dealing with standard features of Java
com.darwinsys.locks	Pessimistic locking API
com.darwinsys.mail	Classes for dealing with email, mainly a convenience class for sending mail
com.darwinsys.model	Sample data models
com.darwinsys.net	Networking
com.darwinsys.notepad	A very simple "notepad" style implemented in Java
com.darwinsys.preso	Presentations
com.darwinsys.reflection	Reflection
com.darwinsys.regex	Regular expression stuff: an REDemo program, a `grep` variant
com.darwinsys.security	Security
com.darwinsys.sql	Classes for dealing with SQL databases
com.darwinsys.swingui	Classes for helping construct and use Swing GUIs
com.darwinsys.swingui.layout	A few interesting LayoutManager implementations
com.darwinsys.tel	Telephony
com.darwinsys.testdata	Test data generators
com.darwinsys.tools	Miscellaneous tools

Package name	Package description
com.darwinsys.unix	Unix helpers
com.darwinsys.util	A few miscellaneous utility classes
com.darwinsys.workflow	BreakNagger, a tool to interrupt your workflow for your own good
com.darwinsys.xml	XML utilities

Many classes from these packages are used as examples in this book. You'll also find that some of the other examples have imports from the com.darwinsys packages.

General notes

Your best bet is to use `git clone` to download a copy of both Git projects and then do a `git pull` every few months to get updates. Alternatively, on the book's catalog page you can download a single intersection subset of both libraries that is made up almost exclusively of files actually used in the book. This archive is made from the sources that are dynamically included in the book at formatting time, so it should reflect exactly the examples you see in the book. But it will not include as many examples as the three individual archives, nor is it guaranteed that everything will compile because of missing dependencies, nor will it get updates. But if all you want is to copy pieces into a project you're working on, this may be the one to get. You can find links to all of these files from my own website for this book; just follow the Downloads link.

The two separate repositories contain multiple self-contained projects with support for building with both Eclipse (Recipe 1.4) and Maven (Recipe 2.4). Note that Maven will automatically fetch a vast array of prerequisite libraries when first invoked on a given project, so be sure you're online on a high-speed internet link. Maven will thus ensure that all prerequisites are installed before building. If you choose to build pieces individually, look in the *pom.xml* file for the list of dependencies. Unfortunately, I will not be able to help you with the control files included in the download if you are using tooling other than Eclipse or Maven.

If you have a version of Java older than Java 21, a few files will not compile. While it's better if you upgrade to a newer Java, if that's not possible, then you can make up exclusion elements in Maven or other tooling to avoid compiling these files.

All my code in the two projects is released under the least-restrictive credit-only license, the two-clause BSD license. If you find it useful, incorporate it into your own software. There is no need to write to ask me for permission; just use it, with credit. If you get rich off it, send me some money.

Most of the command-line examples refer to source files, assuming you are in *src/main/java*, and runnable classes, assuming you are in (or have added to your `CLASSPATH`) the build directory (e.g., usually *target/classes*). This will not be mentioned with each example, as doing so would consume a lot of paper.

Caveat lector

The repos have been in development since 1995. This means that you will likely find some code that is not up to date or that no longer reflects best practices. This is not surprising: any body of code will grow old if any part of it is not actively maintained. (Thus, at this point, I invoke Culture Club's song "Do You Really Want to Hurt Me": "Give me time to realize my crime.") Where advice in the book disagrees with some code you found in the repo, keep this in mind. One practice of Extreme Programming is continuous refactoring, the ability to improve any part of the codebase at any time. Don't be surprised if the code in the online source directory differs from what appears in the book; it is a rare month that I don't make some improvement to the code, and the results are committed and pushed quite often. So if there are differences between what's printed in the book and what you get from GitHub, be glad, not sad, for you'll have received the benefit of hindsight. Also, people can contribute easily on GitHub via pull requests; that's what makes it interesting. If you find a bug or an improvement, do send me a pull request! The consolidated archive on the book's catalog page will not be updated as frequently.

CVS, Subversion, Git, Oh My!

Many distributed version control systems or source code management (SCM) systems are available. There are a few open source SCM systems that have been widely used over the years:

- Concurrent Versions System (CVS)
- Apache Subversion
- Git
- Others that are used in particular niches (e.g., Bazaar, Mercurial) and a few commercial tools (Perforce, ClearCase)

Although each has its advantages and disadvantages, the use of Git in the Linux system build process (and projects based on Linux, such as the Android mobile environment), as well as the availability of sites like *github.com*, *gitlab.com*, and others, give Git a massive momentum over the others. I don't have statistics, but I suspect the

number of projects in Git repositories probably exceeds the others combined. Several well-known organizations using Git are listed on the Git home page.[2]

For this reason, I have migrated all my public projects to GitHub; see my GitHub repository. To download the projects and get updates applied automatically, use Git to download them. Options include the following:

- Use the command-line Git client. If you are on any modern Unix or Linux system (including macOS), Git is either included or available in your ports or packaging or developer tools. On modern MS Windows systems, use `winget install git.git`. Finally, Git can also be downloaded for MS Windows, macOS, and Linux from the Git home page under Downloads.
- All modern IDEs have Git support built in (though IntelliJ doesn't include Git itself; it relies on the command-line Git client, possibly because the main Java implementation `jgit` is owned by Eclipse).
- Numerous standalone GUI clients.
- Even continuous integration servers such as Jenkins (see Recipe 2.15) have plugins available for updating a project with Git (and other popular SCM systems) before building them.

You will want to have one or more of these Git clients at your disposal to download my code examples. You could instead download the code examples as ZIP archive files, but then you wouldn't get updates! You can also view or download individual files from the GitHub page via a web browser.

1.4 Compiling, Running, and Testing with an IDE

Problem

It is cumbersome to use several tools for the various development tasks.

Solution

Use an Integrated Development Environment (IDE), which combines editing, testing, compiling, running, debugging, and package management.

2 Git is not an acronym. The name was chosen by Linux founder Linus Torvalds; he once quipped that he had named the software after himself. Due to Git's popularity, there is already one fairly complete clone called Got, which stands for "Game of Trees", a pun involving source code directory trees and a once-popular TV show.

Discussion

Many programmers find that using a handful of separate tools—a text editor, a compiler, and a runner program, not to mention a debugger—is too many. An IDE *integrates* all of these into a single toolset with a graphical user interface. Many IDEs are available, and the better ones are fully integrated tools with their own copies of the compilers and virtual machines. Class browsers and other features of IDEs round out the ease-of-use feature sets of these tools. Today most developers use an IDE because of the productivity gains. Although I started as a command-line junkie, I do find that IDE features like the following make me more productive:

Code completion
: *Ian's Rule* here is that I never type more than three characters of any name that is known to the IDE; let the computer do the typing!

Incremental compiling features
: Note and report compilation errors as you type, instead of waiting until you are finished typing.

Refactoring
: The ability to make far-reaching yet behavior-preserving changes to a codebase without having to manually edit dozens of individual files.

Beyond that, I don't plan to debate the merits of IDE versus the command-line process; I use both modes at different times and on different projects. I'm just going to show a few examples of using a couple of the Java-based IDEs.

The most popular Java IDEs, which run on all mainstream computing platforms and quite a few niche ones, are *Eclipse*, *IntelliJ IDEA*, *VSCode*, and *NetBeans*. Eclipse was for a long time the most widely used by Java-only developers. Eclipse was created by IBM, which set up the Eclipse Software Foundation to host it; the ESF now hosts dozens of open source projects including JakartaEE, the follow-on to Java Enterprise Edition. The Eclipse Platform is also used as the basis of other tools such as SpringSource Tool Suite (STS) and IBM's Rational Application Developer (RAD). If you develop for Android, the Android tooling was originally based on Eclipse, but transitioned a decade or so ago to IntelliJ as the basis for Android Studio, which is now the standard IDE for Android and for Google's other mobile platform, Flutter.

IntelliJ IDEA is a project of JetBrains, and has both a free community edition and a pro ("Ultimate") edition. Note that JetBrains also provides the JetBrains Toolbox App for macOS, Windows, and Linux. This will install IntelliJ IDEA, Android Studio, PyCharm, and a host of other JetBrains IDEs, and allow you to update all the IDEs that are out-of-date with a single click.

Microsoft's VSCode is a later entrant, originally for .NET developers. VSCode is open source and, like the others, supports a variety of languages in addition to Java.

NetBeans was a startup company and was bought by Sun, which was then bought by Oracle, which in turn donated NetBeans to the Apache Software Foundation.

All these Java IDEs are plug-in based and offer a wide selection of optional and third-party plug-ins to enhance the IDE, such as supporting other programming languages, frameworks, and file types. For Eclipse, use the Eclipse Marketplace, near the bottom of the Help menu. For IntelliJ IDEA, use the Settings → Plugins menu item.

While the following paragraph shows creating and running a program with Eclipse, the IntelliJ IDEA and NetBeans IDEs all offer similar capabilities. All three of these IDEs are written mostly in Java. IntelliJ uses some OS-specific native code, and is also the basis of half a dozen other IntelliJ IDEs including PyCharm, CLion, and others. Eclipse has its own GUI toolkit, unfortunately not written in Java. As a result, on geekly systems like OpenBSD we have IntelliJ IDEA and NetBeans but not Eclipse.

All IDEs do basically the same thing for you when getting started. The example in Figure 1-1 shows starting a new project.

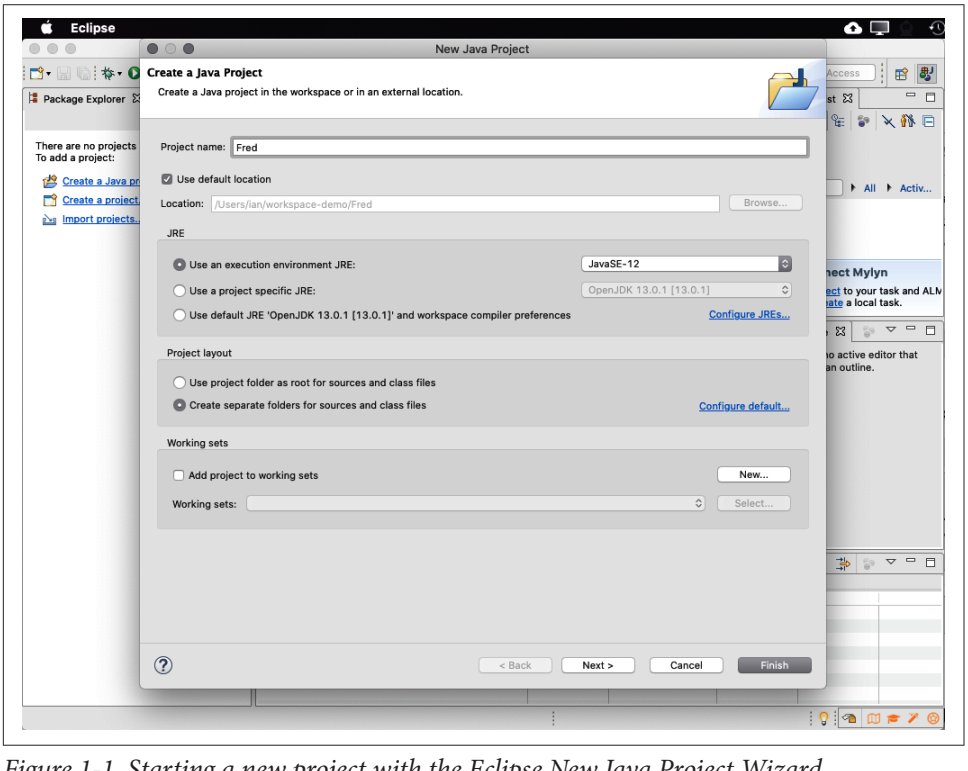


Figure 1-1. Starting a new project with the Eclipse New Java Project Wizard

The Eclipse New Java Class Wizard shown in Figure 1-2 shows creating a new class.

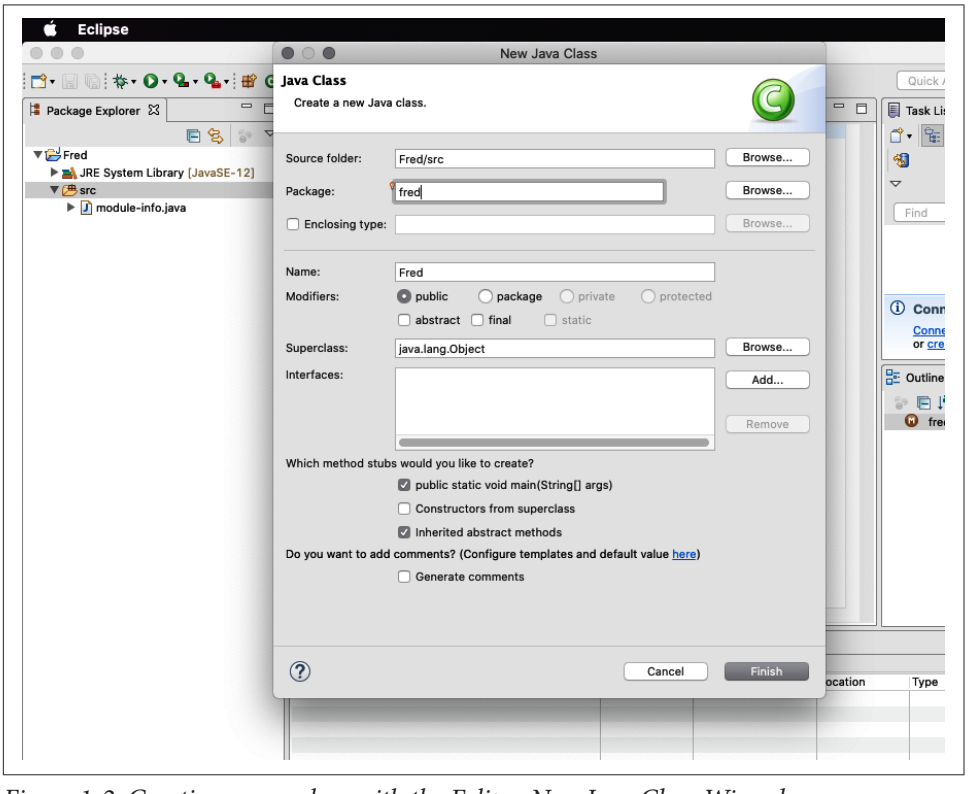


Figure 1-2. Creating a new class with the Eclipse New Java Class Wizard

Eclipse, like all modern IDEs, features a number of refactoring capabilities, shown in Figure 1-3.

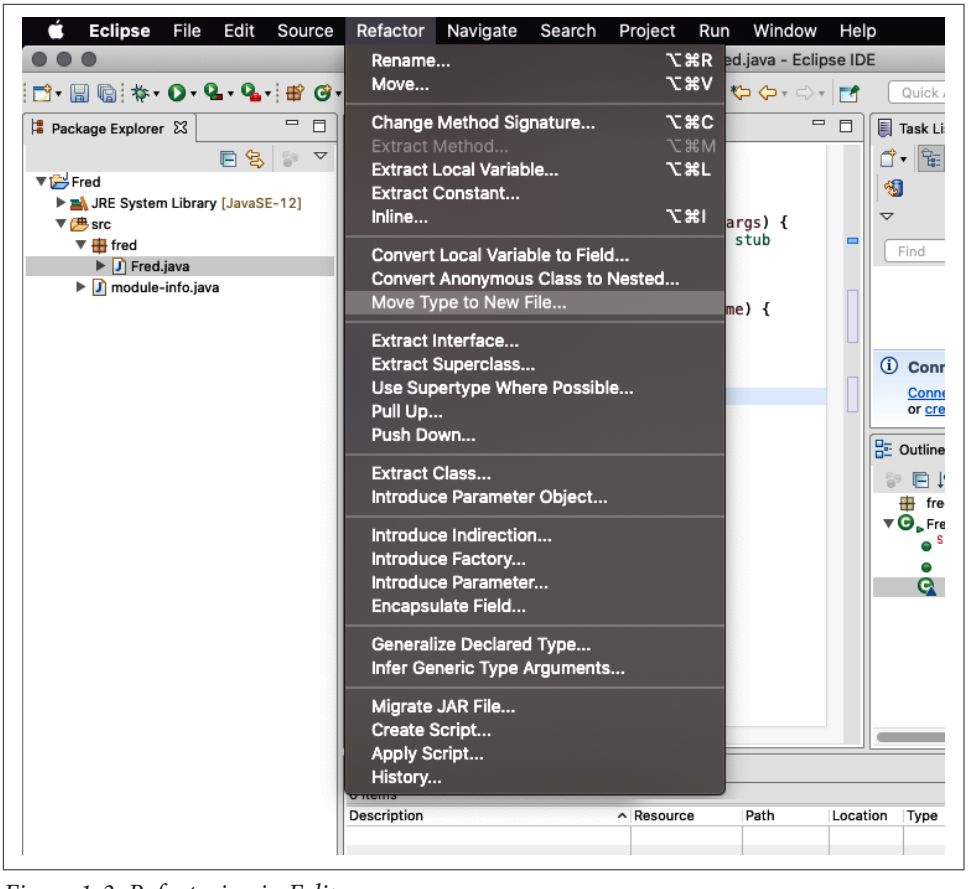


Figure 1-3. Refactoring in Eclipse

And, of course, all the IDEs allow you to run and/or debug your application. Figure 1-4 shows running a Hello, World application using IntelliJ IDEA, and Figure 1-5 shows a similar application in NetBeans.

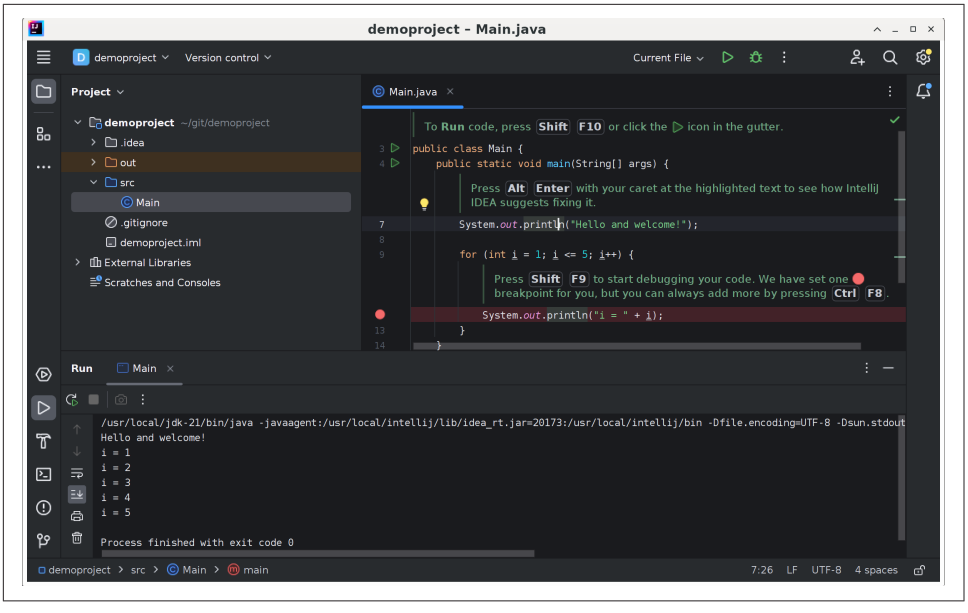


Figure 1-4. IntelliJ run program output

macOS (Release 10.x of the OS) is built upon a BSD Unix (and "Mach") base. As such, it has a regular command line (the Terminal application, hidden away under */Applications/Utilities*) and most of the traditional Unix command-line tools for developers and other users. Mac fans can use one of the many full IDE tools discussed in Recipe 1.4. The main IDE from Apple, Xcode, doesn't really work for Java developers.

Microsoft's open source VSCode has been getting some attention in Java circles lately, but it's not a Java-specific IDE. Give it a try if you like.

How do you choose an IDE? Perhaps it will be dictated by your organization or chosen by majority vote of your fellow developers. Given that all three major IDEs (Eclipse, NetBeans, and IntelliJ) can be downloaded for free, and two of them are open source, why not try them all and see which one best fits the kind of development you do? Regardless of what platform you use to develop Java, there are enough Java IDEs from which to choose. Once you have chosen one, do spend time learning the keyboard shortcuts; they will save you days of time a year if used effectively. Eclipse users should check out my article "Eclipse Shortcuts You Cannot Afford to Ignore".

Note that IntelliJ IDEA puts many "helper" comments on screen that don't appear in the source code being edited.

NetBeans has most of the same capabilities. Figure 1-5 shows a program being run in NetBeans.

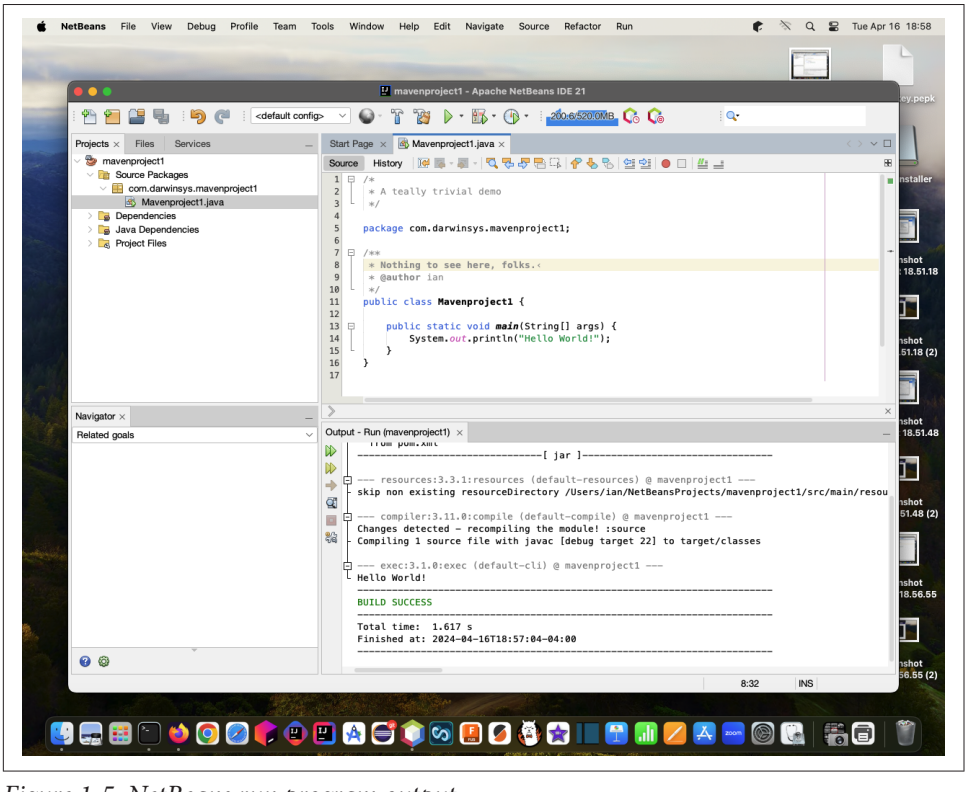


Figure 1-5. NetBeans run program output

See Also

See Table 1-4 for the websites for each major IDE.

Table 1-4. The three major Java IDEs and their websites

Product name	Project URL	Note
Eclipse	*https://eclipse.org*	Basis of STS, RAD
IntelliJ IDEA	*https://jetbrains.com/idea*	Basis of Android Studio
NetBeans	*https://netbeans.apache.org*	Runs anywhere Java SE does

For Eclipse, I have some useful information at my website (*https://darwinsys.com/java*). That site includes a list of shortcuts to aid developer productivity.

1.5 Exploring Java with JShell 11

Problem

You want to try out Java expressions and APIs quickly, without having to create and run a source file every time.

Solution

Use JShell, Java's interactive REPL (read-evaluate-print loop) interpreter.

Discussion

Starting with Java 11, JShell is included as a standard part of the JDK. It allows you to enter Java statements and have them evaluated without the bother of creating a class and a main program. You can use it for quick calculations, to try out an API to see how it works, or for almost any purpose; if you find an expression you like, you can copy it into a regular Java source file and make it permanent. JShell can also be used as a scripting language over Java, but the overhead of starting the JVM means that it won't be as fast as AWK, Perl, or Python for quick scripting.

REPL programs are very convenient, and they are hardly a new idea (LISP languages from the 1950s included them). You can think of command-line interpreters (CLIs) such as the bash or ksh shells on Unix/Linux, or `Command.com` and PowerShell on Microsoft Windows, as REPLs for the system as a whole. Many interpreted languages like Ruby and Python can also be used as REPLs. Java has its own REPL, *JShell*. Here's an example of its use:

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> "Hello"
$1 ==> "Hello"

jshell> System.out.println("Hello");
Hello

jshell> System.out.println($1)
Hello

jshell> "Hello" + sqrt(57)
|  Error:
|  cannot find symbol
|    symbol:   method sqrt(int)
|  "Hello" + sqrt(57)
|            --
```

```
jshell> "Hello" + Math.sqrt(57)
$2 ==> "Hello7.54983443527075"

jshell> String.format("Hello %6.3f", Math.sqrt(57)
   ...> )
$3 ==> "Hello  7.550"

jshell> String x = Math.sqrt(22/7) + " " + Math.PI +
   ...> " and the end."
x ==> "1.7320508075688772 3.141592653589793 and the end."

jshell>
```

You can see some obvious features and benefits here:

- The value of an expression is printed without needing to call `System.out.println` every time, but you can call it if you like.
- Values that are not assigned to a variable get assigned synthetic identifiers, like `$1`, that can be used in subsequent statements.
- The semicolon at the end of a statement is optional (unless you type more than one statement on a line).
- If you omit a close quote, parenthesis, or other punctuation, JShell will just wait for you, giving a continuation prompt (...).
- If you make a mistake, you get a helpful message immediately.
- If you do make a mistake, you can use "shell history" (i.e., up arrow) to recall the statement so you can repair it.
- Not shown but worth knowing: you can get API completion with a single tab, as in shell filename completion.
- You can get the relevant portion of the Javadoc documentation on known classes or methods with just a double tab.

JShell is also useful in prototyping Java code. For example, I wanted one of those health-themed timers that reminds you to get up and move around a bit every half hour:

```
$ jshell
|  Welcome to JShell -- Version 11.0.2
|  For an introduction type: /help intro

jshell> while (true) { sleep (30*60); JOptionPane.showMessageDialog(null,
  "Move it"); }
|  Error:
|  cannot find symbol
|    symbol:   method sleep(int)
|  while (true) { sleep (30*60); JOptionPane.showMessageDialog(null, "Move
it");}
```

```
|                 ^---^
|  Error:
|  cannot find symbol
|    symbol:   variable JOptionPane
|  while (true) { sleep (30*60); JOptionPane.showMessageDialog(null, "Move
it");}
|                                ^---------^

jshell> import javax.swing.*;

[At this point I simply hit up-arrow and return]

jshell> while (true) { Thread.sleep (30*60 * 1000);
  JOptionPane.showMessageDialog(null, "Move it"); }

jshell> ^D
```

I then put the final working version into a Java file called *BreakTimer.java*, put a `class` statement and a `main()` method around the main line, told the IDE to reformat the whole thing, and saved it into my *darwinsys-api* repository.

So go ahead and experiment with JShell. Read the built-in introductory tutorial for more details! When you get something you like, either use `/save`, or copy and paste it into a Java program and save it.

Read more about JShell at the OpenJDK JShell Tutorial.

1.6 Using CLASSPATH Effectively

Problem

You need to keep your class files in a common directory or library, or you're wrestling with `CLASSPATH`.

Solution

Set `CLASSPATH` to the list of directories and/or JAR (Java ARchive) files that contain the classes you want.

Discussion

`CLASSPATH` is a list of class directories containing compiled class files in any of a number of directories or JAR files. Just like the `PATH` your system uses for finding programs, the `CLASSPATH` is used by the Java runtime to find classes. Even when you type something as simple as `java HelloWorld`, the Java interpreter looks in each place named in your `CLASSPATH` until it finds a match. Let's work through an example.

The CLASSPATH can be set as an environment variable the same way you set other environment variables, such as your PATH environment variable. However, it's usually preferable to specify the CLASSPATH for a given command on the command line:

```
C:\> java -classpath c:\ian\classes starting.HelloWorld
```

We can use the -d option of javac to compile classes into such a directory:

```
$ javac -d $HOME/classes HelloWorld.java
$ java -cp $HOME/classes starting.HelloWorld
Hello, world!
```

Suppose your CLASSPATH were set to *C:\classes;.* on Windows or *~/classes:.* on Unix or Mac. Suppose you had just compiled a source file named *HelloWorld.java* (with no package statement) into *HelloWorld.class* in the default directory (which is your current directory) and tried to run it. On Unix, if you run one of the kernel tracing tools (trace, strace, truss, or ktrace), you would probably see the Java program open or stat or access the following files:

- Some file(s) in the JDK directory
- Then *~/classes/HelloWorld.class*, which it probably wouldn't find
- Finally, *./HelloWorld.class*, which it would find, open, and read into memory

The vague "some file(s) in the JDK directory" is release dependent. You should not mess with the JDK files, but if you're curious, you can find them in the system properties (see "Getting information from system properties" on page 43).

The reason I and others suggest *not* setting CLASSPATH as an environment variable is that we don't like surprises. It's easy to add a JAR to your CLASSPATH and then forget that you've done so; a program might then work for you but not for your colleagues, due to their being unaware of your hidden dependency. And if you add a new version to CLASSPATH without removing the old version, you may run into conflicts.

Note also that providing the -classpath argument causes the CLASSPATH environment variable to be ignored.

If you still want to set CLASSPATH as an environment variable, you can. Suppose you had also installed the JAR file containing the supporting classes for programs from this book, *darwinsys-api.jar* (the actual filename if you download it may have a version number as part of the filename). You might then set your CLASSPATH to *C:\classes;C:\classes\darwinsys-api.jar;.* on Windows or *~/classes:~/classes/darwinsys-api.jar:.* on Unix.

Notice that you *do* need to list the full name of the JAR file explicitly. Unlike a single class file, placing a JAR file into a directory listed in your CLASSPATH does not make it available.

While these examples show explicit use of `java` with `-classpath`, it is generally more convenient (and reproducible) to use a build tool such as Maven (Recipe 2.4) or Gradle (Recipe 2.5), which automatically provides the `CLASSPATH` for both compilation and execution.

Note that Java 9 and later also have a module path (environment variable `MODULEPATH`, command-line argument `--module-path entry[:,…]`) with the same syntax as the classpath. The module path contains code that has been modularized; the Java Platform Module System is discussed in Recipes 2.2 and 2.3.

1.7 Documenting Classes with Javadoc

Problem

You have heard about this thing called *code reuse* and would like to promote it by informing developers (including yourself, later) how to use your classes.

Solution

Use Javadoc. Write the comments when you write the code.

Discussion

Javadoc is one of the great inventions of the early Java years. Like so many good things, it was not wholly invented by the Java folks; earlier projects such as Knuth's Literate Programming had combined source code and documentation in a single source file. But the Java folks did a good job on it, and it came along at the right time. Javadoc is to Java classes what *man pages* are to Unix, or what Windows Help is to Windows applications: it is a standard format that everybody expects to find and knows how to use. Learn it (this takes some discipline: learn how others use it effectively). Use it. Write it. Maintain it. Live long and prosper (well, perhaps that's not guaranteed). All that HTML documentation that you learned from writing Java code, the complete reference for the JDK—did you think they hired dozens of tech writers to produce it? Nay, that's not the Java way. Java's developers wrote the documentation comments as they went along, and when the release was made, they ran `javadoc` on all the zillions of public classes and generated the documentation bundle at the same time as the JDK. You can, should, and really must do the same when you are preparing classes for other developers to use.

All you have to do to use Javadoc is to put special *Javadoc comments* into your Java source files. These are similar to multiline Java comments, but they begin with a slash and *two* stars and end with the normal star-slash. Javadoc comments must appear immediately before the definition of the class, method, or field that they document; if placed elsewhere, they are ignored.

A series of keywords, prefixed by the at sign, can appear inside Javadoc comments in certain contexts. Some are contained in braces. The keywords are listed in Table 1-5.

Table 1-5. Javadoc keywords

Keyword	Use
`@author`	Author name(s)
`{@code` *`text`*`}`	Displays text in code font without HTML interpretation
`@deprecated`	Explains deprecation warning (when, why, what to replace with)
`{@docroot}`	Refers to the root of the generated documentation tree
`@exception`	Alias for `@throws`
`{@inheritDoc}`	Inherits documentation from nearest superclass/superinterface
`@link`	Generates inline link to another class or member
`{@link` *`ref`* label`}`	Generates inline link with label to another class or member
`@linkplain`	Same as `@link` but displays in plain text
`{@literal` *`text`*`}`	Displays text without interpretation
`@param` *`name description`*	Argument name and meaning (methods only)
`@return`	Returns value
`@see`	Generates cross-reference link to another class or member
`@serial`	Describes serializable field
`@serialData`	Describes order and types of data in serialized form
`@serialField`	Describes serializable field
`@since`	Tells the JDK version in which introduced (primarily for Sun use)
`@snippet` **18**	Includes code snippet
`@throws`	Exception class and conditions under which thrown
`{@value [`*`ref`*`]}`	Displays values of this or another constant field
`@version`	Version identifier

Example 1-2 is a somewhat contrived example that shows some common Javadoc keywords in use. The output of running this through `javadoc` is shown in a browser in Figure 1-6.

Example 1-2. main/src/main/java/javadoc/JavadocDemo.java

```
/**
 * Construct the GUI
 * @throws java.lang.IllegalArgumentException if constructed on a Sunday.
 */
public JavadocDemo() {
  // We create and add a pushbutton here,
  // but it doesn't do anything yet.
  Button b = new Button("Hello");
```

```
    add(b);            // connect Button into component
    // Totally capricious example of what you should not do
    if (LocalDate.now().getDayOfWeek() == DayOfWeek.SUNDAY) {
      throw new IllegalArgumentException("Never On A Sunday");
    }
  }

  /** paint() is an AWT Component method, called when the
   *  component needs to be painted. This one just draws colored
   * boxes in the window.
   *
   * @param g A java.awt.Graphics that we use for all our
   * drawing methods.
   */
  @Override
  public void paint(Graphics g) {
    // ...
```

The `javadoc` tool works fine for one class but really comes into its own when dealing with a package or collection of packages. You can provide a package summary file for each package, which will be incorporated into the generated files. Javadoc generates thoroughly interlinked and crosslinked documentation, just like that which accompanies the standard JDK. There are numerous command-line options; I normally use `-author` and `-version` to get it to include these items, and often `-link` to tell it where to find the standard JDK to link to. There is also the output directory argument `-d` to avoid cluttering the source folder with the dozens of files that Javadoc creates.

Run `javadoc -help` for a complete list of options, or see the full documentation online at Oracle's website. Figure 1-6 shows one view of the documentation that the class shown in Example 1-2 generates when run as the following:

```
$ javadoc -author -version JavadocDemo.java
```

If you run this with Java 9+, it will also include a fully functional search box, as shown in the upper right of Figure 1-6. This is implemented in JavaScript, so it should work in any modern browser.

Be aware that quite a few files are generated, and one of the generated files will have the same name as each class, with the extension *.html*. Since you probably wish to avoid cluttering up your source directories with the generated files, the `-d` *`directory path`* option for Javadoc allows you to place the generated files into the specified directory.

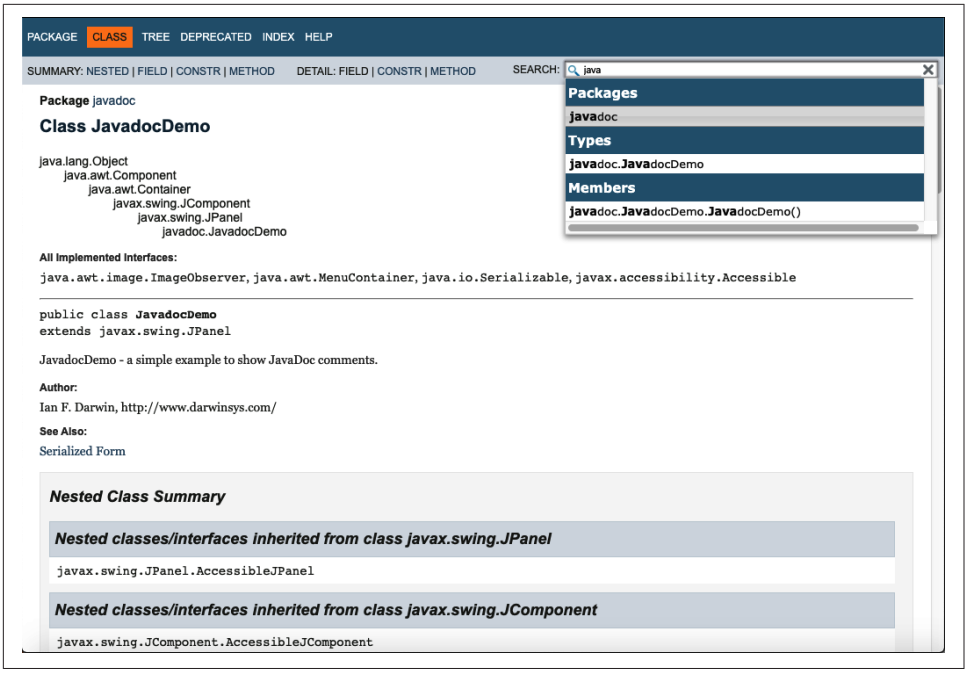


Figure 1-6. Javadoc opened in a browser

Snippets 18

As of Java 18, Javadoc can include internal or external "code snippets" that provide a brief code sample showing use of the class or method being documented. Example 1-3 shows an "internal" snippet, where the sample code is embedded in the `@snippet` tag (the tag and its content is required to be surrounded by a {…} pair).

Example 1-3. main/src/main/java/javadoc/JavadocDemo.java

```
/**
 * A simple demo method. Typical usage:
 * {@snippet lang="java" :
 *  var demo = new JavadocDemo();
 *  demo.demo(42); // or some other int
 * }
 * @param i The value to be processed.
 */
public void demo(int i) {
  System.out.printf("Demo value is %d\n", i);
}
```

When processed via `javadoc`, the result in Figure 1-7 is created. Note the convenient Copy button on the right!

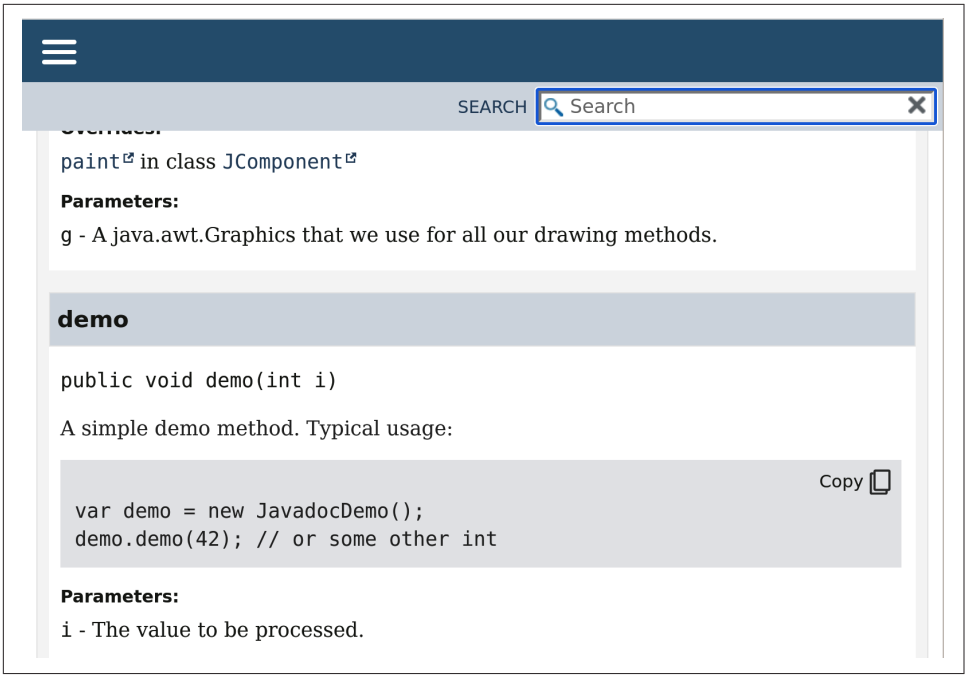


Figure 1-7. An internal snippet

The content of snippets is not limited to Java code; any text format can be used. Internal snippets cannot contain /**...**/ comments, nor an excess of mismatched "}" characters.

External snippets are files that are included in the generated documentation. They can be in a `snippet-files` subdirectory of the directory containing the source code or in an external folder (which must be passed as `--snippet-path` to the `javadoc` command). They can optionally contain subset "regions," delimited by `// @start region=name` and ending with `// @end [region=name]`. If the code is in another language than Java, its comment convention can be used. If the region argument isn't provided on the `@end`, the most recent region is ended. Example 1-4 shows an external snippet.

Example 1-4. main/src/main/java/javadoc/JavadocDemo.java

```
/**
 * A simple method. See this note:
 * {@snippet lang="python" file="snyde_comment.py"}
 * @param i The value to be processed.
 */
public void demo2(int i) {
```

```
    System.out.printf("Demo value is %d\n", i);
}
```

External snippets in the default location (in `snippet-files` under the package source directory) seem only to be found if you run Javadoc from the top-level directory, as is normally done with `mvn javadoc:javadoc`.

There's considerably more capability to the snippet mechanism. For more details please refer to the JEP 413 documentation.

Markdown 23

Java 23 introduced an entirely new mechanism for writing Javadoc, through the use of Markdown, a simple document formatting language. I would have preferred AsciiDoc, but they didn't ask me, so JEP 467 specifies Markdown.

Markdown Javadoc is introduced by a triple slash (`///`) on each line, rather than the traditional `/** ... */` comment. You still use most of the keywords from Table 1-5. However, Markdown simplifies document formatting, so instead of using HTML tags, you can just write text, with empty lines for paragraph breaks, leading dashes for list items, and square brackets (`[]`) for JDK links or `[`*`link text`*`](`*`https://link_url_address`*`)` for external links.

Example 1-5 shows the same code as Example 1-2 but with most of the Javadoc changed to Markdown. In the online copy of this file, one snippet example has been left in the traditional format to show that you can mix and match both formats in the same file.

Example 1-5. main/src/main/java/javadoc/JavadocMdDemo.java

```
///
/// JavadocMdDemo - a MarkDown example to show JavaDoc comments.
///
/// In Java 23, JEP 467 specifies [Markdown](https://en.wikipedia.org/wiki/Markdown)
/// as an alternate implementation language to prepare Javadoc.
///
/// Markdown Javadoc uses three slashes per line instead of the /** ... */ by
/// traditional JavaDoc. This format has some advantages:
/// - it's easier to write, and to read in the code
/// - bullet points like this are easier to format
/// - etc.
///
/// @author Ian Darwin https://darwinsys.com/java
///
public class JavadocMdDemo extends JPanel {
```

```
private static final long serialVersionUID = 1L;

///
/// Construct the GUI
/// @throws java.lang.IllegalArgumentException if constructed on a Sunday.
///
public JavadocMdDemo() {
      // ...
```

When run, it looks like Figure 1-8.

Class JavadocMdDemo
java.lang.Object
java.awt.Component
java.awt.Container
javax.swing.JComponent
javax.swing.JPanel
javadoc.JavadocMdDemo
All Implemented Interfaces:
ImageObserver, MenuContainer, Serializable, Accessible
public class JavadocMdDemo
extends JPanel
JavadocMdDemo - a MarkDown example to show JavaDoc comments.
In Java 23, JEP 467 specifies Markdown as an alternate implementation language to prepare Javadoc.
Markdown Javadoc uses three slashes per line instead of the /** ... */ by traditional JavaDoc. This format has some advantages:
it's easier to write, and to read in the code
bullet points like this are easier to format
etc.
See Also:
Serialized Form
Nested Class Summary
Nested classes/interfaces inherited from class javax.swing.JPanel
JPanel.AccessibleJPanel

Figure 1-8. Javadoc prepared with Markdown

See Also

The `javadoc` tool has numerous other command-line arguments. If documentation is for your own use only and will not be distributed, you can use the `-link` option to tell it where your standard JDK documentation is installed so that links can be generated to standard Java classes (like `String`, `Object`, and so on). If documentation is to be distributed, you can omit `-link` or use `-link` with a URL to the appropriate Java API page on Oracle's website. See the online tools documentation for all the command-line options.

The output that Javadoc generates is fine for most purposes. It is possible to write your own `Doclet` class to make the Javadoc program into a class documentation

verifier, a Java-to-other-format (such as Java-to-AsciiDoc) documentation generator, or whatever you like. See the Javadoc tools documentation that comes with the JDK for documents and examples, or go to Oracle's website.

Javadoc and Dash/Zeal

Dash is a Mac-centric documentation viewer. There is an open source implementation called Zeal, as well as plug-ins for several popular IDEs and editors including the popular Java IDE IntelliJ IDEA. Dash's sponsor Kapeli provides over 200 sets of documents for free download. You can quite easily package a set of Javadoc pages into a docset that will be usable by both these programs; use Kapeli's tool for macOS or this version for other platforms. This is great for internal use. If your software is being made available to the public or to a decent-sized customer base, you may want to publish the docset back to Dash.

Javadoc Versus JavaHelp

Javadoc is for programmers using your classes; for a GUI application, users of desktop applications will appreciate standard online help. This is the role of the JavaHelp API, which is not covered in this book but is fully explained in *Creating Effective JavaHelp* by Kevin Lewis (O'Reilly). JavaHelp is another useful specification that was left to coast during the Sun sellout to Oracle; what remains of it—the old source code—is hosted at javahelp.

1.8 Beyond Javadoc: Annotations/Metadata

Problem

You want to generate not just documentation from your source code, but also other code artifacts. You want to mark code for additional compiler verification.

Solution

Use the Java annotations, or metadata, facility.

Discussion

The continuing success of the open source tool XDoclet—originally used to generate the tedious auxiliary classes and deployment descriptor files for the widely criticized EJB2 framework—led to a demand for a similar mechanism in standard Java. Java *annotations* were the result. The annotation mechanism uses an interface-like syntax, in which both declaration and use of annotations use the name preceded by an at character (`@`). This was chosen, according to the designers, to be reminiscent of "Javadoc tags, a preexisting ad hoc annotation facility in the Java programming language."

Javadoc is ad hoc only in the sense that its `@` tags were never fully integrated into the language; most were ignored by the compiler, but `@deprecated` was always understood by the compiler.

Annotations can be read at runtime by use of the Reflection API; this is discussed in Recipe 17.12, where I also show you how to define your own annotations. Annotations can also be read post–compile time by tools such as code generators (and others to be invented, perhaps by you, gentle reader!).

Annotations are also read by `javac` at compile time to provide extra information to the compiler.

For example, a common coding error is overloading a method when you mean to override it, by mistakenly using the wrong argument type. Consider overriding the `equals` method in `Object`. If you mistakenly write:

```
public boolean equals(MyClass obj) {
    ...
}
```

then you have created a new overload that will likely never be called, and the default version in `Object` will be called. To prevent this, an annotation included in `java.lang` is the `Override` annotation. This has no parameters but is simply placed before the method call, like this:

```
/**
 * AnnotationOverrideDemo - Simple demonstation of Metadata being used to
 * verify that a method does in fact override (not overload) a method
 * from the parent class. This class provides the method.
 */
abstract class Top {
    public abstract void myMethod(Object o);
}

/** Simple demonstration of Metadata being used to verify
 * that a method does in fact override (not overload) a method
 * from the parent class. This class is supposed to do the overriding,
 * but deliberately introduces an error to show how the modern compiler
 * behaves
 */
class Bottom {

    @Override
    public void myMethod(String s) {    // EXPECT COMPILE ERROR
        // Do something here...
    }
}
```

Attempting to compile this results in a compiler error that the method in question does not override a method, even though the annotation says it does; this is a fatal compile-time error:

```
C:> javac AnnotationOverrideDemo.java
AnnotationOverrideDemo.java:16: method does not override a method
            from its superclass
        @Override public void myMethod(String s) {     // EXPECT COMPILE ERROR
         ^
1 error
C:>
```

As it should be.

1.9 Packaging and Running JAR Files

Problem

You want to create a Java archive (JAR) file from your package (or any other collection of files). You want to be able to run your program from the JAR file.

Solution

Use the command-line `jar` tool. Or use a build tool that will do the "jarring" for you. Create the JAR file with a `Main-Class:` entry in its manifest; run the program with the `java -jar` option.

Discussion

The `jar` archiver is Java's standard tool for building archives. Archives serve the same purpose as the program libraries that some other programming languages use. Java normally loads its standard classes from archives, a fact you can verify by running a simple "Hello, World" program with the `-verbose` option:

```
java -verbose HelloWorld
```

Creating an archive is a simple process. The `jar` tool takes several command-line arguments: the most common are `c` for create, `t` for table of contents, and `x` for extract. The archive name is specified with `-f` and a filename. The options are followed by the files and directories to be archived, like this:

```
jar cvf /tmp/MyClasses.jar .
```

The dot at the end is important; it means the current directory. This command creates an archive of all files in the current directory and its subdirectories into the file */tmp/MyClasses.jar*.

Most applications of JAR files depend on an extra file that is always present in a true JAR file, called a *manifest*. This file always lists the contents of the JAR and their attributes; you can add extra information into it. The attributes are in the form `name: value`, as used in email headers, properties files (see Recipe 7.10), and elsewhere. Some attributes are required by the application, whereas others are optional. For example, running a main program directly from a JAR requires a `Main-Program` header. You can even invent your own attributes, such as the following:

```
MySillyAttribute: true
MySillynessLevel: high (6'2")
```

You store this in a file called, say, *manifest.stub*,[3] and pass it to `jar` with the `-m` switch. `jar` includes your attributes in the manifest file it creates:

```
jar -cv -m manifest.stub -f /tmp/com.darwinsys.util.jar .
```

The `jar` program and related tools add additional information to the manifest, including a listing of all the other files included in the archive.

The `java` command has a `-jar` option that tells it to run the main program found within a JAR file. In this case, it will also find classes it needs to load from within the same JAR file. How does it know which class to run? You must tell it. Create a one-line file like this, noting that the attribute fields are case-sensitive and that the colon must be followed by a space:

```
Main-Class: com.somedomainhere.HelloWorld
```

Place that in a file called, say, *manifest.stub*, and assume that you want to run the program `HelloWorld` from the given package. You can then use the following commands to package your app and run it from the JAR file:

```
C:> javac HelloWorld.java
C:> jar cvmf manifest.stub hello.jar HelloWorld.class
C:> java -jar hello.jar
Hello, World of Java
C:>
```

You can now copy the JAR file anywhere and run it the same way. You do not need to add it to your `CLASSPATH` or list the name of the main class.

On GUI platforms that support it, you can launch this application by double-clicking the JAR file. This works on macOS, Microsoft Windows, and most X Windows desktops.

In real life you would probably automate the JAR creation with Maven. Maven will automatically create a JAR file from your source project when you run `mvn package`.

3 Some people like to use names like *MyPackage.mf* so that it's clear which package it is for; the extension *.mf* is arbitrary, but it's a good convention for identifying manifest files.

You can configure the appropriate Maven plug-in with `<manifest>` elements instead of having the information in a separate text file, as is done in the following:

```
<project ...>
    ...
    <packaging>jar</packaging>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.4</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addclasspath>true</addclasspath>
                            <mainClass>${main.class}</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

You have to provide the actual main class, of course, either as a `<property>` or by replacing `${main-class}` in the `<manifest>` section. With this in place, `mvn package` will build a runnable JAR file. However, if your class has external dependencies, the preceding steps will not package them, and you will get a missing class exception when you run it. For this, you need to use the Maven assembly plug-in:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
            <manifest>
                <addDefaultImplementationEntries>true
                </addDefaultImplementationEntries>
                <mainClass>${main.class}</mainClass>
                <!-- <manifestFile>manifest.stub</manifestFile> -->
            </manifest>
            <manifestEntries>
                <Vendor-URL>https://YOURDOMAIN.com/SOME_PATH/</Vendor-URL>
            </manifestEntries>
        </archive>
```

```
        </configuration>
    </plugin>
```

Now the invocation `mvn package assembly:single` will produce a runnable JAR with all dependencies. Note that your *target* folder will contain both *foo-0.0.1-SNAPSHOT.jar* and *foo-0.0.1-SNAPSHOT-jar-with-dependencies.jar*; the latter is the one you need.

The `jpackage` tool (see Recipe 2.18) will do a more complete job of packaging than `assembly:single` and is included with Java 14+.

1.10 Creating a JAR That Supports Multiple Versions of Java

Problem

You are packaging a JAR file, whether a complete application or a library, for others to use. You want to use new Java features in code you release, but still provide backward compatibility for those who for some reason must run with older JDK releases.

Solution

Create a multi-release JAR file.

Discussion

"Always in motion the Java is," according to Yoda. This means devs always get cool new features to play with, or at least, to take advantage of. The multi-release JAR file provides a mechanism for forward compatibility to allow use of new versions of classes utilizing new features, while still allowing your code to run on older Java runtimes. To use this, provide multiple versions of the same source file updated and compiled for various releases. Create an extra directory in the JAR file for each newer-than-legacy version of the code. And you keep the older JDK installation around to compile the legacy version of the code.

This example uses a traditional Java `class` for pre-Java-16 JDKs, and a `record` for Java 16 and later. One requirement is that both must expose exactly the same API so that one can substitute for the other.

Here are the two versions of my trivial demo data object. They both have the same class name and the same list of methods (actually, only one method). Of course the source code must be in two different directories. First, the main program:

```
public class Main {
  public static void main(String[] args) {
    var type = new Data().type();
```

```
    System.out.printf("Run with a Data object of type '%s'.\n", type);
  }
}
```

Then there is a legacy (class) version of `Data`:

```
public class Data {
  String type() {
    return "class";
  }
}
```

Here is a Java 16 (record-based) version of the same `Data` class:

```
public record Data() {
  public String type() {
    return "record";
  }
}
```

It should be clear that either `Data` class will work with the main program. The `getType()` method is only there as a demo to identify which version is being called.

There is a `run_demo` script in the top-level diretory of `$js/multireleasejar`. The script runs the main program against both versions of `Data` without recompiling main, just to show that both work correctly and are interchangeable. The script then sets up the structure for the JAR file as follows:

```
|-- MANIFEST.MF
|-- META-INF
|   |-- MANIFEST.MF
|   +-- versions
|       +-- 16
|           +-- mrjdemo
|               +-- Data.class
+-- mrjdemo
        |-- Data.class
        +-- Main.class
```

The manifest file simply contains one line, `Multi-Release: true`, to inform Java to watch for multiple versions when classloading from the JAR.

Here is the `run_demo` script itself. Note that `withjava` is my custom command to run with a particular release of Java:

```
# run the multi-release-jar demo
D=/tmp/multirelease
mkdir $D
echo Running from directory with simple classpath
withjava 11 javac -d $D *.java
withjava 11 java -cp $D mrjdemo.Main
echo Running from directory with version up front in classpath
javac -d $D/META-INF/versions/16 versions/16/*.java
```

```
java -cp $D/META-INF/versions/16:$D mrjdemo.Main
cp MANIFEST.MF $D
cd $D
jar cvmf MANIFEST.MF /tmp/multirelease.jar mrjdemo META-INF
echo "Running from jar with legacy java"
withjava 11 java -cp /tmp/multirelease.jar mrjdemo.Main
echo "Running from jar with current java"
java -cp /tmp/multirelease.jar mrjdemo.Main
```

Running it produces this output:

```
$ sh run_demo
Running from directory with simple classpath
Run with a Data object of type 'class'.
Running from directory with version up front in classpath
Run with a Data object of type 'record'.
5 directories, 5 files
added manifest
adding: mrjdemo/(in = 0) (out= 0)(stored 0%)
adding: mrjdemo/Data.class(in = 275) (out= 215)(deflated 21%)
adding: mrjdemo/Main.class(in = 575) (out= 370)(deflated 35%)
ignoring entry META-INF/
adding: META-INF/versions/(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/16/(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/16/mrjdemo/(in = 0) (out= 0)(stored 0%)
adding: META-INF/versions/16/mrjdemo/Data.class(in = 1018) (out= 498)(deflated
51%)
Running from jar with legacy java
Run with a Data object of type 'class'.
Running from jar with current java
Run with a Data object of type 'record'.
$
```

The Java runtime correctly uses the `class` version when run on Java 11, and the `record` version when run on Java 16 or later.

The creation of the JAR file can be done a bit more automatically with some command-line options added to the `jar` command as of Java 9. It can also be automated using Maven or Gradle. I did it this "verbose" way to ensure that all the individual steps are visible.

1.11 Packaging Web Tier Components into a WAR File

Problem

You have some web tier resources and want to package them into a single file for deploying to the server.

Solution

Use `jar` to make a web archive (WAR) file. Or, as mentioned earlier, use Maven with `packaging="war"`.

Discussion

Java EE/Jakarta EE defines a series of server-side components for use in web servers. They can be packaged for easy installation into a web server. A *web application* in the Servlet API specification is a collection of HTML/JSP/JSF pages, servlets, and other resources. A typical directory structure might include the following:

```
Project Root Directory
├── README.asciidoc
├── index.html - typical web pages
|── signup.jsp - ditto
└── WEB-INF Server directory
    ├── classes - Directory for individual .class files
    ├── lib    - Directory for Jar files needed by app
    └── web.xml - web app Descriptor ("Configuration file")
```

Once you have prepared the files in this way, you just package them up with a build tool. Using Maven, with `<packaging>war</packaging>`, your tree might look like this:

```
Project Root Directory
├── README.asciidoc
├── pom.xml
└── src
    └── main
        ├── java
        |   └── foo
        |       └── WebTierClass.java
        └── webapp
            ├── WEB-INF
            |   ├── classes
            |   ├── lib
            |   └── web.xml
            ├── index.html
            └── signup.jsp
```

Then `mvn package` will compile things, put them in place, and create the WAR file for you, leaving it under *target*. Gradle users would use a similar directory structure. You then deploy the resulting WAR file into your web server. For details on the deployment step, consult the documentation on the particular server you're using.

1.12 Compiling and Running Java: GraalVM for Better Performance

Problem

You've heard that Graal is a JVM from Oracle that's faster than the standard JDK, and you want to try it out. Graal promises to offer better performance, largely obtained by pre-compiling your Java code into executable form for a given platform. And it offers the ability to mix and match programming languages.

Solution

Download and install GraalVM.

Discussion

GraalVM bills itself as "a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Clojure, Kotlin, and LLVM-based languages such as C and C++."

Note that Graal is undergoing change. There may be newer versions and changed functionality by the time you are ready to install.

You could build your own GraalVM, since the complete source is on GitHub. However, it's easier to download a pre-built binary. Start at the downloads page. You can download a tarball for Linux, macOS, and Windows. There is no formal installer at this point. To install it, open a terminal window and try the following (the directory chosen is for macOS):

```
$ cd /Library/Java/JavaVirtualMachines # or ~/Library/...
# In this, replace 23, macos and aarch64 with your actual download:
$ sudo tar xzvf ~/Downloads/graalvm-jdk-23_macos-aarch64_bin.tar.gz
$ cd
$ /usr/libexec/java_home -V # macOS only
    24 (arm64) "Oracle Corporation" - "OpenJDK 24-ea"
               /Library/Java/JavaVirtualMachines/jdk-24.jdk/Contents/Home
    23.0.1 (arm64) "Oracle Corporation" - "Oracle GraalVM 23.0.1+11.1"
               /Library/Java/JavaVirtualMachines/graalvm-jdk-23.0.1+11.1/
Contents/Home
    23 (arm64) "Oracle Corporation" - "OpenJDK 23-ea"
               /Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home
    22 (arm64) "Oracle Corporation" - "OpenJDK 22-ea"
               /Library/Java/JavaVirtualMachines/jdk-22.jdk/Contents/Home
    17.0.11 (arm64) "JetBrains s.r.o." - "JBR-17.0.11+1-1312.2-nomod 17.0.11"
               /Users/ian/Library/Java/JavaVirtualMachines/jbr-17.0.11/
Contents/Home
    21.0.1 (arm64) "Eclipse Adoptium" - "OpenJDK 21.0.1"
```

```
                /Library/Java/JavaVirtualMachines/temurin-21.jdk/Contents/Home
$
```

macOS does not provide a means to permanently set the default version; you have to adjust `JAVA_HOME` each time you want to change. The `java_home` command output confirms that you have installed GraalVM, but it's not your default JVM yet:

```
export JAVA_HOME=$(/usr/libexec/java_home -v 23.0.1)
```

On most versions of Linux, after installing a JDK, you can use the standard Linux `alternatives` command (e.g., `sudo alternatives --config java`) to make this your default. On other systems, do the install in a sensible place, and set `JAVA_HOME` and/or `PATH` as needed.

Now you should be running the Graal version of Java. This is what you should see:

```
$ java -version
java version "23.0.1" 2024-10-15
Java(TM) SE Runtime Environment Oracle GraalVM 23.0.1+11.1
        (build 23.0.1+11-jvmci-b01)
Java HotSpot(TM) 64-Bit Server VM Oracle GraalVM 23.0.1+11.1
        (build 23.0.1+11-jvmci-b01, mixed mode, sharing)
$
```

Your output will probably differ, but as long as it looks vaguely like this and says "GraalVM" you should be good.

Graal includes a number of useful tools, including `native-image`, which can in some cases translate a class file into a binary executable for the platform it's running on, optimizing startup speed and also reducing the download size needed to run a single application. The `native-image` tool is included in the download. This tool reads the compiled *.class* file, so if you have set your `PATH` correctly, then:

```
$ javac Hello.java
$ native-image Hello
```

should produce a runnable binary. Here's an example. The first command is just to show that Graal generally works like a JDK VM and that it can compile and run a Java program. Then we build and run a native-compiled version:

```
$ java Hello.java
Hello on 2024-10-24
$ javac Hello.java
$ native-image Hello
========================================================================================
GraalVM Native Image: Generating hello (executable)...
========================================================================================
[1/8] Initializing...                                            (23.1s @ 0.07GB)
[2/8] Performing analysis...  []                               (3.5s @ 0.24GB)
[3/8] Building universe...                                      (0.6s @ 0.23GB)
[4/8] Parsing methods...      []                                (0.4s @ 0.24GB)
[5/8] Inlining methods...     []                               (0.4s @ 0.26GB)
```

```
[6/8] Compiling methods...    []                              (7.5s @ 0.23GB)
[7/8] Laying out methods...   []                                (0.5s @ 0.27GB)
[8/8] Creating image...       []                              (0.9s @ 0.23GB)
---------------------------------------------------------------------------
Build artifacts:
 /tmp/hello (executable)
===========================================================================
============================
Finished generating hello in 37.3s.
$ ./hello
Hello on 2024-10-24
$
```

The `native-image` tool does take longer to create its output, and it is *extremely* verbose, both compared to `javac` (about 30 lines had to be elided from the preceding printout to keep it short enough). There is an option, `--silent`, to make `native-image` behave like `javac` in this regard, but it's worth running without this option the first time. After that, you may wish to `alias native-image="native-image --silent"` (or write a batch file or script if you're using Windows `cmd` or PowerShell).

You can also use Graal to mix-and-match languages like Python and JavaScript. We'll explore calling some other languages from Java in Recipe 18.4.

See the GraalVM website for more information on GraalVM.

1.13 Getting Information About the Environment, OS, and Runtime

Problem

You need to know how your Java program can deal with its immediate surroundings with what we call the runtime environment. In one sense, everything you do in a Java program using almost any Java API involves the environment. This recipe covers environment variables, system properties, and operating-system dependencies.

Solution

Use methods of the `System` and `Runtime` classes, which know a lot about your particular system.

Discussion

There are several locations from which to obtain this information.

Getting environment variables

Use `System.getenv()`.

The seventh edition of Unix, released in 1979, had a then-new feature known as environment variables. Environment variables are in all modern Unix systems (including macOS) and in most later command-line systems, such as the DOS or Command Prompt in Windows. Environment variables are commonly used for customizing an individual computer user's runtime environment, hence the name. To take one familiar example, on Unix or DOS the environment variable `PATH` determines where the system looks for executable programs. So, of course developers want to know how to access environment variables from their Java program.

The answer is that you can do this in all modern versions of Java. In some ancient versions of Java, `System.getenv()` was deprecated and/or just didn't work. Nowadays the `getenv()` method is no longer deprecated, though it still carries the warning that system properties (see "Getting information from system properties" on page 43) should be used instead. Even among systems that support environment variables, their names are case-sensitive on some platforms and case-insensitive on others. The code in Example 1-6 is a short program that uses the `getenv()` method. Note that it is spelled `getenv()`, not the expected `getEnv()`—and it's probably too old to change.

Example 1-6. main/src/main/java/environ/GetEnv.java

```
public class GetEnv {
  public static void main(String[] argv) {
    String envVarName = argv.length == 0 ? "PATH" : argv[0];
    System.out.printf("System.getenv(\"%s\") = %s\n",
      envVarName, System.getenv(envVarName));
  }
}
```

Running this code on Windows will produce output similar to the following, with your actual `PATH` instead of one I had a while ago:

```
System.getenv("PATH") = C:\windows\bin;c:\jdk-17\bin;c:\Users\ian\bin
```

On Unix (where most nonsystem programs such as Java are installed in */usr/local/bin*), the code is:

```
System.getenv("PATH") = /home/ian/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/
local/bin
```

The no-argument form of the method `System.getenv()` returns *all* the environment variables in the form of an immutable `String Map`. You can iterate through this map and access all the user's settings or retrieve multiple environment settings.

Getting information from system properties

To get information from the system properties, use `System.getProperty()` or `System.getProperties()`.

What is a *property* anyway? A property is just a name and value pair stored in a `java.util.Properties` object, which we discuss more fully in Recipe 7.10.

The `System.Properties` object controls and describes the Java runtime. The `System` class has a static `Properties` member whose content is the merger of operating system specifics (`os.name`, for example), system and user tailoring (`java.class.path`), and properties defined on the command line (as we'll see in a moment). Note that the use of periods in the names of system properties (like `os.arch`, `os.version`, `java.class.path`, and `java.lang.version`) makes it look as though there is a hierarchical relationship similar to that for package/class names. The `Properties` class, however, imposes no such relationships: each key is just a string, and dots are not special.

To view all the defined system properties, you can iterate through the output of calling `System.getProperties()` as in Example 1-7.

Example 1-7. `jshell System.getProperties()`

```
jshell> System.getProperties().forEach((k,v) -> System.out.println(k + "->" +v))
awt.toolkit->sun.awt.X11.XToolkit
java.specification.version->21
sun.cpu.isalist->
sun.jnu.encoding->UTF-8
java.class.path->.
java.vm.vendor->Oracle Corporation
sun.arch.data.model->64
java.vendor.url->https://java.oracle.com/
user.timezone->
os.name->OpenBSD
java.vm.specification.version->21
... many more ...
jshell>
```

Note that properties whose names begin with `sun` are unsupported and subject to change. There used to be a variable named `sun.boot.class.path`, which contained a list of the archives used at JVM startup. But that entry is not found anymore. Let's look for any property with `boot` in its name:

```
jshell> System.getProperties().forEach((k,v) -> {
 ... if (((String)k).contains("boot")) System.out.println(k + "->" +v);})
sun.boot.library.path->/usr/local/jdk-21/lib
```

So a boot library path is there (the location will vary on different OSes), but no other `boot` entries. As stated, "subject to change."

To retrieve one system-provided property, use `System.getProperty(propName)`. If I wanted to find out if the system properties had a property named `"pencil_color"`, I could say:

```
String sysColor = System.getProperty("pencil_color");
```

But what does that return? Surely Java isn't clever enough to know about everybody's favorite pencil color? Right you are! But we can easily tell Java about our pencil color (or anything else we want to tell it) using the `-D` argument.

When starting a Java command, define a value in the `System.Properties` object just by using a `-D` option. Its argument must have a name, an equals sign, and a value, which are parsed the same way as in a properties file (see Recipe 7.10). You can have more than one `-D` definition between the `java` command and your class name on the command line. At the Unix or Windows command line, type:

```
$ java -D"pencil_color=Deep Sea Green" environ.SysPropDemo
```

Here's the result:

```
-- listing properties --
java.specification.version=21
jdk.internal.javac.source=21
sun.jnu.encoding=UTF-8
java.class.path=
pencil_color=Deep Sea Green
java.vm.vendor=Oracle Corporation
... many more ...
```

When running this under an IDE, put the variable's name and value in the appropriate dialog box, for example, in Eclipse's Run Configuration dialog under Program Arguments. You can also set environment variables and system properties using the build tools (Maven, Gradle, etc.).

The `SysPropDemo` program has code to extract just one or a few properties, so you can run it like this:

```
$ java environ.SysPropDemo os.arch
os.arch = x86
```

If you invoke the `SysPropDemo` program with no arguments, it outputs the same information as the `jshell` fragment in Example 1-7.

Which reminds me—this is a good time to mention system-dependent code. "Dealing with code that depends on the Java version or the operating system" on page 45 talks about OS-dependent code and release-dependent code.

Recipe 7.10 lists more details on using your own `Properties` files. The Javadoc page for `java.util.Properties` lists the exact rules used in the `load()` method, as well as other details.

Dealing with code that depends on the Java version or the operating system

In the very rare case when you need to write code that adapts to the underlying operating system, use `System.Properties` to find out the Java version and the operating system, or use the `Files` and `Path` classes to check whether certain system-dependent files or directories are present. You can also use `java.awt.TaskBar` to see if you can use the system-dependent taskbar or dock, and `java.awt.Desktop` to see if you can open a file in the default browser, for example.

Some things depend on the version of Java you are running. Use `System.getProperty()` with an argument of `java.specification.version` to find out the Java version.

Alternatively, and with greater generality, you may want to test for the presence or absence of particular classes. One way to do this is with `Class.forName("class")` (see Chapter 17), which throws an exception if the class cannot be loaded—a good indication that it's not present in the runtime's library. Example 1-8 shows code for this, from an application wanting to find out whether the common Swing UI components are available. The Javadoc for the standard classes reports the version of the JDK in which this class first appeared, under the heading "Since." If there is no such heading, it normally means that the class has been present since the beginnings of Java.

Another alternative for dealing with different Java versions is the multi-release JAR file, covered in Recipe 1.10.

Example 1-8. main/src/main/java/starting/CheckForSwing.java

```
public class CheckForSwing {
  public static void main(String[] args) {
    try {
      Class.forName("javax.swing.JButton");
    } catch (ClassNotFoundException e) {
      String failure =
        "Sorry, but this version of MyApp needs \n" +
        "a Java Runtime with javax.swing GUI components.\n" +
        "Please check your Java installation and try again.";
      // Better to make something appear in the GUI. Either a
      // Dialog, or: myPanel.add(new Label(failure));
      // Can't be JDialog as Swing isn't present.
      System.err.println(failure);
    }
    // No need to print anything here - the GUI should work...
```

```
  }
}
```

It's important to distinguish between testing this code at compile time and at runtime. In both cases, it must be compiled on a system that includes the classes you are testing for: in this case, JDK >= 1.1 and Swing, respectively. These tests are only attempts to help the poor backwater Java runtime user trying to run your up-to-date application. The goal is to provide this user with a message more meaningful than the simple "class not found" error that the runtime gives. Put the test early in the main flow of your application, before any GUI objects are constructed. Otherwise the code just sits there wasting space on newer runtimes and never gets run on Java systems that don't include Swing. Obviously this is a very early example, but you can use the same technique to test for any runtime feature added at any stage of Java's evolution (see Appendix A for an outline of the features added in each release of Java). You can also use this technique to determine whether a needed third-party library has been successfully added to your `CLASSPATH`. You can also use the Reflection API to determine if a class has a given method; see Chapter 17.

Also, although Java is designed to be portable, some things aren't. These include such variables as the filename separator. Everybody on Unix knows that the filename separator is a slash character (/) and that a backward slash, or backslash (\), is an escape character. Back in the late 1970s, a group at Microsoft was actually working on Unix—their version was called Xenix, later taken over by SCO—and the people working on DOS saw and liked the Unix filesystem model. The earliest versions of MS-DOS didn't have directories; it just had user numbers like the system it was a quick and dirty clone of, Digital Research CP/M (itself using ideas from various other systems). So the Microsoft developers set out to clone the Unix filesystem organization. Unfortunately, MS-DOS had already committed the real slash character for use as an option delimiter, for which Unix had used a dash (`-`); and the `PATH` separator (:) was also used as a drive letter delimiter, as in `C:` or `A:`. So we now have commands like those shown in Table 1-6.[4]

Table 1-6. Directory listing commands

System	Directory list command	Meaning	Example `PATH` setting
Unix	`ls -R /`	Recursive listing of /, the top-level directory	`PATH=/bin:/usr/bin`
DOS	`dir /s \`	Directory with subdirectories option (i.e., recursive) of \, the top-level directory (but only of the current drive)	`PATH=C:\windows;D:\mybin`

4 People only exposed to MS-Windows tend to sometimes refer to the character \ as "slash" instead of "backslash" or to write URLs with \ instead of /. Ah well.

Where does this get us? If we are going to generate filenames in Java, we may need to know whether to put a / or a \ or some other character. Java has two solutions to this. First, when moving between Unix and Microsoft systems, at least, it is *permissive*: either / or \ can be used,[5] and the code that deals with the operating system sorts it out. Second, and more generally, Java makes the platform-specific information available in a platform-independent way. For the file separator (and also the `PATH` separator), the `java.io.File` class makes available some static variables containing this information. Because the `File` class manages platform-dependent information, it makes sense to anchor this information here. The variables are shown in Table 1-7.

Table 1-7. Filename properties

Name	Type	Meaning
`separator`	`static String`	The system-dependent filename separator character (e.g., / or \), as a string for convenience
`separatorChar`	`static char`	The system-dependent filename separator character (e.g., / or \)
`pathSeparator`	`static String`	The system-dependent path separator character, as a string for convenience
`pathSeparatorChar`	`static char`	The system-dependent path separator character

Both filename and path separators are normally characters, but they are also available in `String` form for convenience.

You can use the `System.properties` object to determine the operating system you are running on. We previously showed some code that simply lists the system properties; it can be informative to run that on several different Java implementations on different OSes.

Some OSes, for example, provide a mechanism called the null device that can be used to discard output (typically used for timing purposes, or to discard standard output while retaining system error output). Here is code that asks the system properties for the `os.name` and uses it to make up a name that can be used for discarding data (if no null device is known for the given platform, we return the name `jnk`, which means that on such platforms, we'll occasionally create, well, junk files; I just remove these files when I stumble across them):

```
package com.darwinsys.lang;

import java.io.File;

/** Some things that are system dependent.
```

5 When compiling strings with backslashes for use on Windows, remember to double them because \ is an escape character in most places other than the MS-DOS command line: `String rootDir = "C:\\";`.

```
 * All methods are static.
 * @author Ian Darwin
 */
public class SysDep {

    final static String UNIX_NULL_DEV = "/dev/null";
    final static String WINDOWS_NULL_DEV = "NUL:";
    final static String FAKE_NULL_DEV = "jnk";

    /** Return the name of the "Null Device" on platforms which support it,
     * or "jnk" (to create an obviously well-named temp file) otherwise.
    * @return The name to use for output.
     */
    public static String getDevNull() {

        if (new File(UNIX_NULL_DEV).exists()) {      ❶
            return UNIX_NULL_DEV;
        }

        String sys = System.getProperty("os.name"); ❷
        if (sys==null) {                             ❸
            return FAKE_NULL_DEV;
        }
        if (sys.startsWith("Windows")) {             ❹
            return WINDOWS_NULL_DEV;
        }
        return FAKE_NULL_DEV;                        ❺
    }
}
```

❶ If `/dev/null` exists, use it.

❷ If not, ask `System properties` if it knows the OS name.

❸ Nope, so give up, return `jnk`.

❹ We know it's Microsoft Windows, so use `NUL:`.

❺ All else fails, go with `jnk`.

CHAPTER 2

Software Development, Testing, and Maintenance

2.0 Introduction

This chapter focuses on techniques for building larger applications, maintaining code quality, and ensuring code correctness. First up is coverage of both Apache Maven and Gradle Inc.'s eponmyous Gradle, the two main build tools that download and cache dependencies, build applications, and do much more for you in terms of automation. The remaining recipes concentrate on *testing* and *maintaining* software effectively.

2.1 Designing Applications: Packages, Modules

Problem

You want to be able to import classes and/or organize your classes, so you want to create your own package. You want to design software with a good structure that will work with the Java Module System.

Solution

Evaluate the major sections of your application, and consider its structure as one or more major modules, and packages within modules. Put a `package` statement at the front of each file, and recompile with `-d` or a build tool or IDE.

Discussion

One of the better aspects of the Java language is that it has defined a very clear packaging mechanism for categorizing and managing its large API. Modules consist of one or more packages, packages consist of classes, and classes consist of methods and fields.

As discussed in Recipe 2.2, modules provide a high degree of separation among different parts of your application. You might have three "tiers" in your application: the user interface, the middle tier or "business" tier, and the backend or data storage tier. These are good candidates for an initial set of modules. Each might initially consist of only one package, but it's good to put the module structure in place at the beginning.

Anybody can create a package, with one important restriction: you and I cannot create a package whose name begins with the four letters `java`. Packages named java. or javax. are reserved for use by Oracle's Java team and *you may not create packages starting with those names*. This is a legal obligation that you accepted when you clicked the clickwrap license agreement when you downloaded the JDK. When Java was new, there were about a dozen packages in a structure that is very much still with us, though it has quadrupled in size; some of these packages are shown in Table 2-1.

Table 2-1. Java packages basic structure

Name	Function
java.awt	Graphical user interface
java.io	Reading and writing
java.lang	Intrinsic classes (`String`, etc.)
java.lang.annotation	Library support for annotation processing
java.math	Math library
java.net	Networking (sockets)
java.nio	"New" I/O (not new anymore): channel-based I/O
java.sql	Java database connectivity
java.text	Handling and formatting/parsing dates, numbers, messages
java.time	Java 8: modern date/time API (JSR-311)
java.util	Utilities (collections, date)
java.util.regex	Regular expressions
javax.naming	JNDI
javax.print	Support for printing
javax.script	Java 6: scripting engines support
javax.swing	Modern graphical user interface

Many packages have been added over the years, but the initial structure has stood the test of time fairly well.

To create your own packages, or to add a class into an existing package, the `package` statement must be the very first noncomment statement in your Java source file—preceding even `import` statements—and it must give the full name of the package. Your package names are expected to start with your domain name backward; for example, my internet domain is *darwinsys.com*, so most of my packages begin with "com.darwinsys" and a project name. The utility classes used in this book and meant for reuse are in one of the com.darwinsys packages listed in Recipe 1.3, and each source file begins with a statement, such as this:

```
package com.darwinsys.util;
```

The demonstration classes in the *javasrc* repository do not follow this pattern. They are in packages with names related to the chapter they are in or the java.* package they relate to; for example, "lang" for basic Java stuff, "structure" for examples from the data structuring chapter (Chapter 7), "threads" for the threading chapter (Chapter 11), and so on. It is hoped that you will put them in a "real" package if you reuse them in your application!

Once you have package statements in place, be aware that the Java runtime, and even the compiler, will expect the compiled *.class* files to be found in their rightful place (i.e., in the subdirectory corresponding to the full name somewhere in your `CLASSPATH` settings). For example, the class file for `com.darwinsys.util.FileIO` must *not* be in the file *FileIO.class* in my `CLASSPATH` but must be in *com/darwinsys/util/FileIO.class* relative to one of the directories or archives in my `CLASSPATH`. Accordingly, if you are compiling with the command-line compiler, it is customary (almost mandatory) to use the `-d` command-line argument when compiling. This argument must be followed by the name of an existing directory (often . is used to signify the current directory) to specify where to build the directory tree. For example, to compile all the *.java* files in the current directory, and create the directory path under it (e.g., create *./com/darwinsys/util* in the example), use this:

```
javac -d . *.java
```

This creates the path (e.g., *com/darwinsys/util/*) relative to the current directory and puts the class files into that subdirectory. This makes life easy for subsequent compilations and also for creating archives, which is covered in Recipe 1.9.

Of course, if you use a build tool such as Maven or Gradle (see Recipe 2.4), this will be done correctly by default, so you won't have to remember to keep doing it!

Note that in all modern Java environments, classes that do not belong to a package (the anonymous package) cannot be listed in an `import` statement, although they can be referred to by other classes in that package. They also cannot become part of a JPMS module (see Recipe 2.2).

2.2 Using the Java Modules System

Problem

You are using Java 9 or later and need to deal with the Modules mechanism.

Solution

Consider adding a *module-info.java* file to relate your code to other modules.

Discussion

Java's Modules system, formerly known as Project Jigsaw, was designed to handle the need to build large applications—particularly the JDK itself—out of many small pieces. To an extent this problem had been solved by build tools like Maven (Recipe 2.4) and Gradle (Recipe 2.5), but the Modules system solves a slightly different problem than those tools. Maven or Gradle will find dependencies, download them, install them on your development and test runtime `CLASSPATH`, and package them into runnable JAR files. The Modules system is more concerned with the visbility of classes from one chunk of application code to another, typically provided by different developers who may not know or trust each other. As such, it is an admission that Java's original set of access modifiers—such as `public`, `private`, `protected`, and default visibility—was just not quite sufficient for building large-scale applications.

What follows is a brief discussion of using the Java Platform Module System (JPMS) to import modules into your application. There is an introduction to creating your own modules in Recipe 2.1. For a more detailed presentation, you should refer to a book-length treatment such as *Java 9 Modularity* by Sander Mak and Paul Bakker (O'Reilly).

Java has always been a language for large-scale development. Object orientation is one of the keys: classes and objects group methods, and access modifiers can be applied so that public and private methods are clearly separated. When developing large applications, having just a single flat namespace of classes is still not enough. Enter packages: they gather classes into logical groups within their own namespace. Access control can be applied at the package level as well so that some classes are only accessible inside a package. Modules are the next logical step up. A *module* groups some number of related packages, has a distinct name, and can restrict access to some packages while exposing other packages to different modules as part of its public API.

One thing to understand at the outset: JPMS is not a replacement for your existing build tool. Whether you provide JAR files by using Maven, Gradle, Ant, or just dump all needed JAR files into a lib directory, you still need to provide them. Also, don't confuse Maven's modules with JPMS modules; the former is the physical structuring of a project into subprojects, and the latter is something the Java platform (compiler,

runtime) understands. Usually when working with Java modules, each Java module will equate to a single Maven module. There's no requirement for this; it's just a technique for retaining your sanity.

When you're dealing with a tiny, self-contained program, you don't need to be concerned with modules. Just put all the necessary JAR files on your `CLASSPATH` at compile time and runtime, and all will be well. Probably.

On some mid-life releases (Java 11 to 18), you may see warning messages like this along the way:

```
Illegal reflective access by com.foo.Bar
        (file:/Users/ian/.m2/repository/com/foo/1.3.1/foo-1.3.1.jar)
        to field java.util.Properties.defaults
Please consider reporting this to the maintainers of com.foo.Bar
Use --illegal-access=warn to enable warnings of further
 illegal reflective access operations
All illegal access operations will be denied in a future release
```

The warning message comes about as a result of JPMS doing its job, checking that no types are accessed in encapsulated packages within a module. Such messages should go away over time as all public Java libraries and all apps being developed get modularized. Meanwhile, if you see these messages, consider upgrading to a newer version of the library.

Probably?

Why will all be well only "probably"? If you are using certain classes that were deprecated over the last few releases, things won't compile. For that, you must make the requisite modules available. In the *javasrc/unsafe* subdirectory (also a Maven module), there is a class called `LoadAverage`. The load average is a feature of Unix/Linux systems that gives a rough measure of system load or busy-ness, by reporting the number of processes that are waiting to be run. There are almost always more processes running than CPU cores to run them on, so some always have to wait. Higher numbers mean a busier system with slower response.

Sun's unsupported `Unsafe` class has a method for obtaining the load average, on systems that support it. Your code has to use the Reflection API (see Chapter 17) to obtain the `Unsafe` object; if you try to instantiate `Unsafe` directly you will get a `SecurityException` (this was the case before the Modules system). Once the instance is obtained and casted to `Unsafe`, you can invoke methods such as `loadAverage()` (Example 2-1).

Example 2-1. unsafe/src/main/java/unsafe/LoadAverage.java (use of Unsafe.java)

```
public class LoadAverage {
  public static void main(String[] args) throws Exception {
    Field f = Unsafe.class.getDeclaredField("theUnsafe");
    f.setAccessible(true);
    Unsafe unsafe = (Unsafe) f.get(null);
    int numElements = 3;
    double loadAverage[] = new double[numElements];
    unsafe.getLoadAverage(loadAverage, numElements);
    for (double d : loadAverage) {
      System.out.printf("%4.2f ", d);
    }
    System.out.println();
  }
}
```

I'll show the compilation of this code shortly; but first, some configuration. If the app is using Java Modules, the *module-info.java* file has to tell the compiler and VM that the app requires use of the module with the semi-obvious name `jdk.unsupported`:

```
    module javasrc.unsafe {
        requires jdk.unsupported;
            // others...
    }
```

I'll say more about the module file format in Recipe 2.3.

Now that we have the code in place and the module file in the top level of the source folder, we can build the project, run the program, and compare its output against the system-level tool for displaying the load average, `uptime`. We'll still get the "internal proprietary API" warnings, but it works:

```
    $ java -version
    openjdk version "21.0.2" 2024-01-16
    OpenJDK Runtime Environment (build 21.0.2+13-1)
    OpenJDK 64-Bit Server VM (build 21.0.2+13-1, mixed mode, sharing)
    $ mvn clean package
    [INFO] Scanning for projects...
    [INFO]
    [INFO] --------------------< com.darwinsys:javasrc-unsafe >--------------------
    [INFO] Building javasrc - Unsafe 1.0.0-SNAPSHOT
    [INFO] --------------------------------[ jar ]---------------------------------
    [INFO]
    [INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ javasrc-unsafe ---
    [INFO] Deleting /Users/ian/workspace/javasrc/unsafe/target
    [INFO]
    [INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ javasrc-
    unsafe ---
    [INFO] Using UTF-8 encoding to copy filtered resources.
    [INFO] skip non existing resourceDirectory /Users/ian/workspace/javasrc/
    unsafe/src/main/resources
```

```
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ javasrc-unsafe
---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /Users/ian/workspace/javasrc/unsafe/target/
classes
[WARNING] /Users/ian/workspace/javasrc/unsafe/src/main/java/unsafe/LoadAver-
age.java:[3,16] sun.misc.Unsafe is internal proprietary API and may be removed
in a future release
...
... multiple similar Java warnings and non-informative Maven messages elided ...
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ javasrc-unsafe ---
[INFO] Building jar: /Users/ian/workspace/javasrc/unsafe/target/javasrc-
unsafe-1.0.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.668 s
[INFO] Finished at: 2020-01-05T14:53:55-05:00
[INFO] ------------------------------------------------------------------------
$
$ java -cp target/classes unsafe/LoadAverage
3.54 1.94 1.62
$ uptime # Just to compare against the standard user-level tool
14:54  up 1 day, 21:50, 5 users, load averages: 3.54 1.94 1.62
$
```

This gives the same numbers as the standard Unix `uptime` command when run on Java 11 through 21. As the warnings imply, it *may* (i.e., probably will) be removed "someday."

If you are building a more complex app, you will probably need to put together a more complete *module-info.java* file. But at this stage it's primarily a matter of requiring the modules you need. The standard Java API is divided into several modules, which you can list using the `java` command:

```
$ java --list-modules
java.base
java.compiler
java.datatransfer
java.desktop
java.instrument
java.logging
java.management
java.management.rmi
java.naming
java.net.http
java.prefs
java.rmi
java.scripting
java.se
```

```
java.security.jgss
java.security.sasl
java.smartcardio
java.sql
java.sql.rowset
java.transaction.xa
java.xml
java.xml.crypto
... plus a bunch of JDK modules ...
```

Of these, `java.base` is always available and doesn't need to be listed in your module file, `java.desktop` adds AWT (the older Abstract Windowing Toolkit) and Swing for graphics, and `java.se` includes basically all of what used to be public API in the Java SDK. If our load average program wanted to display the result in a Swing window, for example, it would need to add this into its module file:

```
requires java.desktop;
```

When your application is big enough to be divided into tiers or layers, you will probably want to describe these modules using JPMS, that is, create your own modules. That is the topic of Recipe 2.3.

2.3 Using JPMS to Create a Module

Problem

You want your packaged archive to work smoothly with the JPMS.

Solution

Create a *module-info.java* file in the root of the source directory.

Discussion

The file *module-info.java* was introduced in Java 9 to provide the compiler and tools with information about your library's needs and what it provides. Note that this is not a valid Java class filename because it contains a minus sign. The module also has a group of pseudokeywords, which only have their special meaning inside a module file. The simplest *module-info* is the following:

```
module foo {
    // Empty
}
```

But just as a Java class with no members won't get you very far in the real world, neither will this empty module file. We need to provide some additional information. For this example, I show how I modularized my `darwinsys-api`, a collection of 150 or so randomly accumulated classes that I reuse sometimes. Remember that Jigsaw

(the module system's early name) was initially proposed as a way of modularizing the overgrown JDK itself. Most applications will need the module `java.base` (which is always included). If they need AWT, Swing, or certain other desktop-application-related classes, they also need `java.desktop`. Thus I add the following line into the module definition:

```
require java.desktop;
```

My code also has some JUnit-annotated classes and makes use of the JavaMail API, so we need those as well. JUnit, however, is only needed at test time. While Maven offers `scope`s for compile, test, and runtime, the Modules system does not.

Unfortunately, as of this writing, there are still some libraries that are not modularized. Fortunately, there is a feature known as *automatic modules*, by which if you place a JAR file on the module path that doesn't declare a module, its JAR filename will be used as the basis of an automatically generated module. At one point, it seemed there wasn't a modularized JavaMail API. At that point, I added the following:

```
require mail
```

Unfortunately, when this code was compiled, Maven's Java compiler module spit out this scary-looking warning:

```
[WARNING] ************************************************************************
[WARNING] * Required filename-based automodules detected. Please don't publish
            this project to a public artifact repository! *
[WARNING] ************************************************************************
```

However, I have since found a modularized version of each API that's needed.

The *module-info* also lists any packages that your module desires to make available, that is, its public API. So we need a series of export commands:

```
exports com.darwinsys.calendar;
exports com.darwinsys.csv;
exports com.darwinsys.database;
...
```

By default, packages that are exported cannot be examined using the Reflection API. To allow a module to introspect (use the Reflection API) on another, say, a domain model used with the Java Persistence API (JPA), use `opens`.

For example, if we have a library that needs to use Reflection on a class in `java.lang`, as indicated by the error messages when running the app, we can try to run the app again with this command-line flag:

```
java --add-opens=java.base/java.lang=ALL-UNNAMED packagename.ClassName
```

One of the purposes of Java interfaces is to allow multiple implementations of a service. This is supported in JPMS by the *service* feature. Where an API is defined as one or more interfaces in one module, and multiple implementations are provided, each

in its own module, the implementation module(s) can define an implementation using `provides ... with`, as in the following:

```
exports com.darwinsys.locks;
provides com.darwinsys.locks.PessimisticLockManager
        with com.darwinsys.locks.PessimisticLockManagerImpl;
```

A module wanting to import my lock interface feature would need a `requires com.darwinsys` and might do something like this in code:

```
import java.util.ServiceLoader;
import java.util.Optional;

Optional<LockManager> opt = ServiceLoader.load(LockManager.class).findFirst();
if (!opt.isPresent()) {
    throw new RuntimeException("Could not find implementation of LockManager");
}
LockManager mgr = opt.get();
```

The use of `findFirst()` reflects the knowledge that there is only one implementation. A more complex module might offer multiple alternatives, which would require iterating on the result of the `load()` call to find the one you wanted. The `Optional` interface is described in Recipe 8.10.

The completed *module-info* for the `darwinsys-api` module is shown in Example 2-2.

Example 2-2. DarwinSys-API module-info

```
module com.darwinsys.api {

  requires transitive java.desktop;
  requires java.net.http;
  requires transitive java.prefs;
  requires transitive java.sql;
  requires java.sql.rowset;
  requires static jakarta.mail;
  requires java.xml;

  exports com.darwinsys.calendar;
  exports com.darwinsys.csv;
  exports com.darwinsys.database;
  exports com.darwinsys.diff;
  exports com.darwinsys.formatting;
  exports com.darwinsys.genericui;
  exports com.darwinsys.geo;
  exports com.darwinsys.graphics;
  exports com.darwinsys.html;
  exports com.darwinsys.io;
  opens   com.darwinsys.io;
  exports com.darwinsys.lang;
  exports com.darwinsys.locks;
```

```
  provides com.darwinsys.locks.PessimisticLockManager
    with com.darwinsys.locks.PessimisticLockManagerImpl;
  exports com.darwinsys.mail;
  exports com.darwinsys.net;
  exports com.darwinsys.notepad;
  exports com.darwinsys.numbers;
  opens   com.darwinsys.numbers;
  exports com.darwinsys.preso;
  exports com.darwinsys.reflection;
  exports com.darwinsys.regex;
  exports com.darwinsys.security;
  exports com.darwinsys.sql;
  exports com.darwinsys.swingui;
  opens   com.darwinsys.swingui;
  exports com.darwinsys.tel;
  exports com.darwinsys.testdata;
  exports com.darwinsys.tools;
  exports com.darwinsys.unix;
  exports com.darwinsys.util;
  exports com.darwinsys.xml;
}
```

See Also

JPMS is no longer new, but some library providers are still learning to use it properly. An early posting was *https://openjdk.java.net/projects/jigsaw/quick-start*. A plan for migrating to modules can be found on Jacob Jenkov's website. A discussion about preparing a multimodule Maven application is offered by Baeldung. *Java 9 Modularity* by Mak and Bakker is probably the most comprehensive treatment of JPMS.

Build Tools: Maven and Gradle

Maven and Gradle are the two most widely used build tools for Java developers. Both help automate the build process and manage dependencies, but they have some differences that may help you decide which to use. Both are JVM-based, giving them a performance advantage over anything that has to run several separate Java processes, like a shell-script-based build system, use of the older Make tool, etc. There is detailed information on Maven in Recipe 2.4 and on Gradle in Recipe 2.5.

Maven and Gradle automatically find all the **.java* files in and under *src/main/java*, compile them, find all the source files in *src/test/java*, run the tests, and create a packaged JAR/WAR/etc. file. They will stop if any step along the way fails, for example, if even a single test fails.

Both tools handle dependency management. You simply list, in the tool's config file, the name and version of each library that you need, and the build tool finds it (usually on Maven Central), downloads it over the internet, saves it in a cache folder for future

use, and adds it to your `CLASSPATH` at the right time—all without your having to write any rules.

The name and version are in three parts, actually: a group ID (normally identifying the organization behind the software), the artifact ID (name of this library), and a version number (usually with three parts, like 10.0.1 for the first release of the 10th major version). These three facts together are called the *coordinates*. In Maven they are written out in XML, and in Gradle they are written on one line with colons between. In most of the book, we give the coordinates for libraries being referenced in the Gradle form, as it's more concise.

Both tools also have special knowledge of `CLASSPATH`, making it easy to set a `CLASSPATH` in various ways for compile time. Maven offers a "scope" of tests for classes and other files that will be on your `CLASSPATH` only when running tests, for example.

The Maven build tool "follows conventions over configurations," that is, it provides sensible defaults (but so does Gradle). Maven uses an XML-based configuration file named *pom.xml* to define the project's settings and dependencies. Since I still primarily use Maven, there are POM files in each of the main projects of the *javasrc* repository. Maven provides a standard project structure (which Gradle mostly adopted, for compatibility) and "lifecycle phases" such as compile, test, and package. This makes it easy for developers to get started with new projects. However, Maven build files are verbose and can be inflexible and difficult to customize.

Gradle is a newer, more flexible build tool that uses either a Groovy- or a Kotlin-based domain-specific language (DSL, also called "little language") for defining project settings and tasks. (Only the Groovy variant is used in this book, as it is the "traditional" choice for Gradle.) Gradle allows for more customization and flexibility in defining build processes, making it a good choice for complex projects. Gradle also has better support for incremental builds and parallel execution, which can improve build performance for large projects. Gradle's popularity increased when it became the only build tool supported by Android Studio for building Android applications.

In summary, Maven is a good choice for developers who prefer convention over configuration and want a simple and easy-to-use build tool. Gradle, on the other hand, is better suited for developers who need more flexibility and customization in their build processes. Ultimately, the choice between Maven and Gradle will depend on the specific requirements of the project and the preferences of the development team.

2.4 Automating Compilation, Testing, and Deployment with Apache Maven

Problem

You want a tool that does it all automatically: downloads your dependencies, compiles your code, compiles and runs your tests, packages the app, and installs or deploys it.

Solution

Use Apache Maven.

Discussion

Maven is a Java-centric build tool that includes a sophisticated, distributed dependency management system that also gives it rules for building application packages such as JAR, WAR, and EAR files and deploying them to an array of different targets. Whereas older build tools focus on the *how*, Maven files focus on the *what*, specifying what you want done.

Maven is controlled by a file called *pom.xml* (for Project Object Model). A sample *pom.xml* might look like this:

```
<project xmlns="https://maven.apache.org/POM/4.0.0"
  xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
                      https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-se-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>my-se-project</name>
  <url>https://com.example/</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <junit5.version>5.8.1</junit5.version>
  </properties>

  <dependencies>
    <dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
```

```
            <version>${junit5.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit5.version}</version>
            <scope>test</scope>
        </dependency>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

This specifies a project called *my-se-project* (my standard-edition project) that will be packaged into a JAR file; it depends on the JUnit 5.x framework for unit testing (see Recipe 2.9) but only needs it for compiling and running tests. If I type `mvn install` in the directory with this POM, Maven will ensure that it has a copy of the given version of JUnit. Not just JUnit, but also anything that JUnit depends on, and anything these in turn depend on—known as *transitive dependencies*. Then it will compile everything (setting `CLASSPATH` to exclude JUnit for the main code and including it when compiling and running tests), run all the unit tests, and if they all pass, generate a JAR file for the program and install the JAR file in my personal Maven repo (under *~/.m2/repository*). This will allow other Maven projects to depend on my new project JAR file.

Note that I haven't had to tell Maven where the main and test source files live, nor how to compile them, nor where to install the JAR—this is all handled by sensible defaults, based on a well-defined project structure. The program source is expected to be found in *src/main/java*, and the tests in *src/test/java*; if it's a web application, the web root is expected to be in *src/main/webapp* by default. Of course, you can override these settings.

Note that even the preceding config file does not have to be written by hand; Maven's archetype generation rules let it build the starting version of any of several hundred types of projects. Here is how to create such a file, along with the standard directory structure and a trivial Hello, World app with a test:

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DgroupId=com.example -DartifactId=my-se-project

[INFO] Scanning for projects...
Downloading: https://repo1.maven.org/maven2/org/apache/maven/plugins/
    maven-deploy-plugin/2.5/maven-deploy-plugin-2.5.pom
[several dozen or hundred lines of downloading POM files and Jar files...]
[INFO] Generating project in Interactive mode
[INFO] Archetype [org.apache.maven.archetypes:maven-archetype-quickstart:1.1]
```

```
    found in catalog remote
[INFO] Using property: groupId = com.example
[INFO] Using property: artifactId = my-se-project
Define value for property version:  1.0-SNAPSHOT: : <press enter>
[INFO] Using property: package = com.example
Confirm properties configuration:
groupId: com.example
artifactId: my-se-project
version: 1.0-SNAPSHOT
package: com.example
 Y: : y
[INFO] ----------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype:
    maven-archetype-quickstart:1.1
[INFO] ----------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.example
[INFO] Parameter: packageName, Value: com.example
[INFO] Parameter: package, Value: com.example
[INFO] Parameter: artifactId, Value: my-se-project
[INFO] Parameter: basedir, Value: /private/tmp
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /private/tmp/
    my-se-project
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6:38.051s
[INFO] Finished at: Sun Jan 06 19:19:18 EST 2018
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------------------
```

Alternatively, you can use `mvn archetype:generate` and select the default from a rather long list of choices. The default is a quickstart Java archetype, which makes it easy to get started.

The IDEs (see Recipe 1.4) have support for Maven. For example, if you use Eclipse, M2Eclipse (m2e) is an Eclipse plug-in that will build your Eclipse project dependencies from your POM file; this plug-in ships with current builds of Eclipse.

A POM file can redefine any of the standard goals. Common Maven goals (predefined by default to do something sensible) include the following:

`clean`
: Removes all generated artifacts

`compile`
: Compiles all source files

`test`
: Compiles and runs all unit tests

`package`
: Builds the package

`install`
: Installs *pom.xml* and the package into your local Maven repository for use by your other projects

`deploy`
: Tries to install the package (e.g., on an application server)

Most of the steps implicitly invoke the previous ones. For example, `package` will compile any missing *.class* files, and run the tests if that hasn't already been done in this run.

To run a CLI or desktop program, without having to set this up specifically in the *pom.xml* file, you can simply use the following:

```
$ mvn exec:java -Dexec.mainClass=com.example.YourClass
```

There are application server–specific targets provided by the app server vendors. As a single example, with the WildFly application server you would install some additional plug-in(s) as per their documentation, then deploy to the app server using the following:

```
mvn wildfly:deploy
```

instead of the regular deploy. Since I use this Maven incantation frequently, I have a shell alias or batch file `mwd` to automate even that.

Maven pros and cons

Maven can handle complex projects and is very configurable. I built the *darwinsys-api* and *javasrc* projects with Maven and let it handle finding dependencies, making the download of the project source code smaller (actually, moving the download overhead to the servers of the projects themselves). The downsides to Maven are that its verbose XML format makes authoring and maintaining the build files harder, and there are fewer shortcuts for common behavior. And Maven can be harder to diagnose when things go wrong. A good web search engine is your friend when things fail.

See Also

Start at the Apache Maven website.

Maven Central: Mapping the World of Java Software

There is an immense collection of software freely available to Maven and Gradle users just for adding a `<dependency>` element or Maven artifact into your *pom.xml* or a dependency into your *build.gradle*. You can search this repository on Maven Central or the Nexus Repository Manager.

Figure 2-1 shows a search for my *darwinsys-api* project and the information it reveals. Note that the `dependency` information listed there is all you need to have the library added to your Maven project; just copy the Dependency Information section and paste it into the `<dependencies>` of your POM, and you're done! Because Maven Central has become the definitive place to look for software, many other Java build tools piggyback on Maven Central. To accommodate these users, in turn, Maven Central offers up the dependency information in a form that half a dozen other build tools can directly use in the same copy-and-paste fashion.

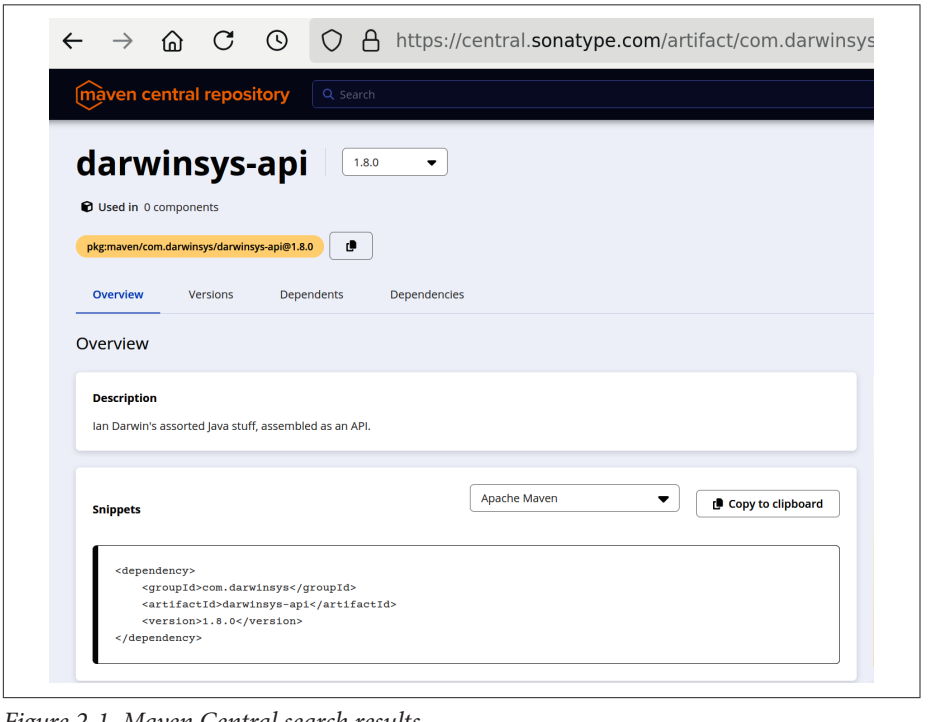


Figure 2-1. Maven Central search results

When you get to the stage of having a useful open source project that others can build upon, you may, in turn, want to share it on Maven Central. The process is longer than building for yourself but not onerous. Refer to this Maven guide or the Sonatype OSS Maven Repository Usage Guide.

One issue I fear is that a bad actor could gain access to a project's site and modify, or install a new version of, a POM. Maven automatically fetches updated POM versions. However, it does use hash signatures to verify that files have not been tampered with during the download process, and all files uploaded to Maven Central must be signed with PGP/GPG, so an attacker would have to compromise both the upload account and the signing keys.

2.5 Automating Compilation, Testing, and Deployment with Gradle

Problem

You want a build tool that doesn't make you use a lot of XML in your configuration file.

Solution

Use Gradle's simple build file format with configuration by convention for shorter build files and fast builds.

Discussion

Gradle is the latest in the succession of build tools (Make, Ant, and Maven). Gradle bills itself as "the enterprise automation tool" and has integration with the other build tools and IDEs.

Unlike Java-based tools Ant and Maven, Gradle doesn't use XML as its scripting language. Instead, it uses one of two DSLs based on the JVM-based and Java-based languages Groovy and Kotlin. To keep the examples simpler, we only cover using Groovy. You don't need to learn Groovy to use Gradle for most projects, only for those that require logic beyond the basics.

On many systems, you can install Gradle from your system's package manager. If not, you can download the latest stable version from the Gradle website, unpack the ZIP file, and add the resulting *bin* subdirectory to your path.

Then you can begin to use Gradle. If you are starting a new project from scratch, navigate into a new, empty directory and use:

```
gradle init
```

This will ask a series of questions such as what type of app to build, what to call it, what language to use (hopefully Java!) and what version thereof, what unit testing framework to use, and so on. The script will go on to create the standard directory structure common to Maven and Gradle. However, it will automatically put all that into a subdirectory called *app*, leaving just a *settings.gradle* to refer to the subdirectory with:

```
rootProject.name = 'gradledemo'
include('app')
```

The intention behind this is to allow you to build a large app out of additional modules, each of which would have its own directory and be included in the settings file.

The `init` script will also create *gradlew.bat* and *gradlew* files, the former for running Gradle under Microsoft Windows and the latter for all other operating systems. Many documents refer to running these, but if you have installed Gradle from your system's package manager, you can continue to invoke this build tool under the name `gradle`.

The *app* subdirectory will contain the main configuration file, *build.gradle*, shown in Example 2-3. The generated file will be in *app/build*. The entire generated app, along with a script of my `init` run, can be found in the *gradledemo* sudirectory of the *javasrc* repository.

For an existing project, assuming you have used the standard source directory (*src/main/java*, *src/main/test*) that is shared by Maven and Gradle, you can adapt the example *build.gradle* file in Example 2-3. You can then use it to build your app, run your unit tests, and, if tests passed and assuming it's a program rather than a library, run the application just by typing the following:

```
gradle run
```

Example 2-3. build.gradle file made by `gradle init`

```
/*
 * This file was generated by the Gradle 'init' task.
 */

plugins {
    // Apply the application plug-in to add support for building a CLI application
in Java.
    id 'application'
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
```

```
    implementation 'com.google.guava:guava:32.1.1-jre'
}

testing {
    suites {
        // Configure the built-in test suite
        test {
            // Use JUnit Jupiter test framework
            useJUnitJupiter('5.9.3')
        }
    }
}

// Apply a specific Java toolchain to ease working on different environments.
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(21)
    }
}

application {
    // Define the main class for the application.
    mainClass = 'com.example.gradledemo.App'
}
```

While the Gradle folks once tried to set up a competitor to Maven Central, the `repositories` line now lets Gradle "borrow" the industry's vast investment in Maven infrastructure.

See Also

As with Maven, there is much more functionality in Gradle. Start at Gradle's website, and see the documentation, in particular Gradle for Java and JVM projects.

2.6 Automating Dependency Management with Maven and Gradle

Problem

You need to manage your application's dependencies upon a range of third-party libraries.

Solution

Automate your dependency management with Maven or Gradle.

Discussion

As any application grows, it will need to depend on third-party libraries for some of its functionality. Those for Java are usually made available in the form of JAR files. In the early days, we would simply collect all the JAR files into one folder and write a batch script to add them all to the `CLASSPATH`. This is neither scalable nor maintainable. In today's software world, it's necessary to use tooling to manage these dependencies. Proper use of tooling will provide the following:

- Documentation on what libraries are being used
- Automatic downloading of these libraries, and verification that they are authentic
- Automatic fetching of updated versions of the libraries as fixes are made to them

Each library is described by giving its "coordinates"—the group (organization), the library name, and usually a version.

The default download site is Maven Central; running your own download site is possible but out of scope. You can find artifacts by name at Maven Central; see Figure 2-1.

Occasionally, software authors will run their own repository, in which case additional configuration will be needed—just follow their documentation.

Maven

For Apache Maven, the dependencies are listed in the `<dependencies>` section of the *pom.xml* file.

Here are some examples:

```
<project ...>
        <groupId>com.example</groupId>
        <artifactId>example_app</artifactId>
        <version>1.0.6</version>
        <packaging>jar</packaging>
        <inceptionYear>2019</inceptionYear>

        <properties>
                ...
                <darwinsys-api-version>1.9.0-SNAPSHOT</darwinsys-api-version>
                <pdfbox-version>2.0.24</pdfbox-version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>org.apache.pdfbox</groupId>
                        <artifactId>pdfbox</artifactId>
                        <version>${pdfbox-version}</version>
                </dependency>
```

```
            <dependency>
                    <groupId>com.darwinsys</groupId>
                    <artifactId>darwinsys-api</artifactId>
                    <version>${darwinsys-api-version}</version>
            </dependency>
            ...
        </dependencies>
    </project>
```

It's quite common for most libraries to have three levels of version number; the leftmost being for major revisions and the rightmost for the smallest change. These versions must be listed in `<version>` tags in the configuration file. It is somewhat common to list the version numbers of libraries in the `<properties>` section and refer to these properties in the `<version>` field, as done here. Where there are several modules from the same project, this avoids repetition and reduces the chance of missing one when updating manually.

When we build the project the first time, the two libraries listed, along with all of their dependencies, will be downloaded. Maven optionally allows a dependency to have a "scope" (where it is visible). Scope values include compile, test, and provided. The first is only on `CLASSPATH` during the build/compile phase. The second is only during testing, and the third is also compile-only, but makes clear that the artifact will be provided at runtime by a container such as a Jakarta EE server or Spring Boot.

To see what versions of these dependencies and their subdependencies are in use, you can run the following:

```
mvn dependency:tree
```

This will print all the dependencies, in a format like this:

```
[INFO] org.example:demo_application:jar:1.0.6
[INFO] +- org.apache.pdfbox:pdfbox:jar:2.0.24:compile
[INFO] |  +- org.apache.pdfbox:fontbox:jar:2.0.24:compile
[INFO] |  \- commons-logging:commons-logging:jar:1.2:compile
```

While we only listed `pdfbox`, `pdfbox` depends on `fontbox`, so that is also downloaded. `pdfbox` also requires `commons-logging`.

After working on the software for some time, we want to update to new versions of the libraries as they become available. This is important in today's internet with many bad actors attempting to exploit unpatched vulnerabilities! If the version numbers are in `<properties>`, use:

```
mvn versions:update-properties
```

If instead the version numbers have been listed explicitly in the `<version>` elements, use:

```
mvn versions:use-latest-versions
```

There are other options for both the `dependency` and `versions` plug-ins; each will accept `help` as its argument, to list all the possibilities.

Gradle

You can easily add dependencies to a Gradle project. Many IDEs have support for doing so. Or you can just open the *build.gradle* file and add lines with the "coordinates" into the `dependencies` section (adding that section if it's missing). For example, here we've added the darwinsys-api library:

```
dependencies {
        implementation 'com.google.guava:guava:32.1.1-jre'
        implementation 'com.darwinsys:darwinsys-api:1.9.0-SNAPSHOT'
}
```

The next time you build the project, the JAR file for that library will be downloaded and cached. Unlike Maven, Gradle does not by default show such downloads:

```
$ vi app/build.gradle
$ gradle run
> Task :app:compileJava
> Task :app:processResources NO-SOURCE
> Task :app:classes

> Task :app:run
Hello World!

BUILD SUCCESSFUL in 2s
2 actionable tasks: 2 executed
$
```

You can view the dependency tree with `gradle dependencies`, but you either have to `cd` down into the *app* subdirectory, or use `gradle :app:dependencies`. This is because the *build.gradle* file is in the *app* subdirectory, and, unlike the `run` task, the `dependencies` task doesn't follow the setting that tells it to look in the *app* module. The Gradle version of this output is substantially longer than the Maven version; here's part of it:

```
$ gradle :app:dependencies

> Task :app:dependencies

------------------------------------------------------------
Project ':app'
------------------------------------------------------------

annotationProcessor - Annotation processors and their dependencies for source
set 'main'.
No dependencies

compileClasspath - Compile classpath for source set 'main'.
```

```
+--- com.google.guava:guava:32.1.1-jre
|    +--- com.google.guava:guava-parent:32.1.1-jre
|    |    +--- com.google.code.findbugs:jsr305:3.0.2 (c)
|    |    +--- org.checkerframework:checker-qual:3.33.0 (c)
|    |    +--- com.google.errorprone:error_prone_annotations:2.18.0 (c)
|    |    +--- com.google.j2objc:j2objc-annotations:2.8 (c)
|    |    \--- junit:junit:4.13.2 (c)
|    +--- com.google.guava:failureaccess:1.0.1
|    +--- com.google.code.findbugs:jsr305 -> 3.0.2
|    +--- org.checkerframework:checker-qual -> 3.33.0
|    +--- com.google.errorprone:error_prone_annotations -> 2.18.0
|    \--- com.google.j2objc:j2objc-annotations -> 2.8
\--- com.darwinsys:darwinsys-api:1.8.0
     +--- org.junit.jupiter:junit-jupiter-engine:5.7.1
     |    +--- org.junit:junit-bom:5.7.1
     |    |    +--- org.junit.jupiter:junit-jupiter-api:5.7.1 (c)
     |    |    +--- org.junit.jupiter:junit-jupiter-engine:5.7.1 (c)
     |    |    +--- org.junit.platform:junit-platform-engine:1.7.1 (c)
     |    |    +--- org.junit.vintage:junit-vintage-engine:5.7.1 (c)
     |    |    \--- org.junit.platform:junit-platform-commons:1.7.1 (c)
...
... many lines elided
...
(c) - A dependency constraint, not a dependency. The dependency affected by the
constraint occurs elsewhere in the tree.
(*) - Indicates repeated occurrences of a transitive dependency subtree...
(n) - A dependency or dependency configuration that cannot be resolved.

A web-based, searchable dependency report is available by adding the --scan
option.

BUILD SUCCESSFUL in 556ms
1 actionable task: 1 executed
$
```

The `n` indicator can be helpful in debugging dependency failure issues.

Gradle does not yet appear to provide a dependency update plug-in. There is an open source plug-in from Ben Manes on GitHub that will report on dependencies needing updates. The easiest way to run it is to add this line into the `plugins` section of *build.gradle*:

```
id "com.github.ben-manes.versions" version "0.51.0"
```

Of course a later version may be available by the time you read this (if so, the plug-in will tell on itself!). After adding this line, you can run a report; for example:

```
$ gradle gradle :app:dependencyUpdates
------------------------------------------------------------
:app Project Dependency Updates (report to plain text file)
------------------------------------------------------------

The following dependencies are using the latest milestone version:
```

```
 - com.github.ben-manes.versions:com.github.ben-manes.versions.gradle.plugin:
0.51.0

The following dependencies have later milestone versions:
 - com.darwinsys:darwinsys-api [1.5.0 -> 1.8.0]
     http://darwinsys.com/darwinsys-api/
 - com.google.guava:guava [32.1.1-jre -> 33.3.1-jre]
     https://github.com/google/guava
...
```

Unlike Maven, this will *not* actually update your *build.gradle* file; you have to do that manually (though some IDEs make it slightly easier).

2.7 Dealing with Deprecation Warnings

Problem

Your code used to compile cleanly, but now it gives deprecation warnings.

Solution

You must have blinked. Either live—dangerously—with the warnings or revise your code to eliminate them.

Discussion

Each new release of Java includes a lot of powerful new functionality, but at a price: during the evolution of these new features, Java's maintainers find some old components that were not designed well or not implemented effectively and should no longer be used because they cannot be properly fixed. In the first major revision, for example, they realized that the `java.util.Date` class had some serious limitations with regard to internationalization. Accordingly, many of the `Date` class methods and constructors are marked "deprecated." According to the *American Heritage Dictionary*, to deprecate something means to "express disapproval of; deplore." Java's developers are therefore disapproving of the old way of doing things. Try compiling this code:

```
import java.util.Date;

/** Demonstrate deprecation warning */
public class Deprec {

  public static void main(String[] av) {

    // Create a Date object for June 6, 2014
    // @SuppressWarnings("deprecation") // Uncomment to suppress
    // EXPECT DEPRECATION WARNING without @SuppressWarnings
    Date d = new Date(2014-1900, 6-1, 10);
```

```
        // Shows why java.time.LocalDate.of(2014,06,10) is safer!
        System.out.println("Date is " + d);
    }
}
```

What happened? When I compiled it (prior to adding the `@SuppressWarnings()` annotation), I got this warning:

```
C:\> javac Deprec.java
Note: Deprec.java uses or overrides a deprecated API.  Recompile with
"-deprecation" for details.
1 warning
C:\>
```

So, we follow orders. For details, recompile with `-deprecation` to see the additional details:

```
C:\> javac -deprecation Deprec.java
Deprec.java:10: warning: constructor Date(int,int,int) in class java.util.Date
has been deprecated
       Date d = new Date(2014-1900, 06-1, 10);    // June 10, 2014
                        ^
1 warning

C:\>
```

The warning is simple: the `Date` constructor that takes three integer arguments has been deprecated. How do you fix it? As in most questions of usage, the answer is to refer to the Javadoc documentation for the class. Of course, the older `Date` class has been replaced by `LocalDate` and `LocalDateTime` (see Chapter 6), so you'd only see that particular example in legacy code, but the principles of dealing with deprecation warnings matter, because some new releases of Java add deprecation warnings to parts of the API that were previously OK to use.

As a general rule, when something has been deprecated, you should not use it in any new code; and, when maintaining code, strive to eliminate the deprecation warnings.

In addition to `Date`, the main areas of deprecation warnings in the standard API are the really ancient event handling and some methods (a few of them important) in the `Thread` class.

You can also deprecate your own code when you come up with a better way of doing things. Put an `@Deprecated` annotation immediately before the class or method you wish to deprecate and/or use an `@deprecated` tag in a Javadoc comment (see Recipe 1.7). The Javadoc comment allows you to explain the deprecation.

See Also

Numerous other tools perform extra checking on your Java code. See my Checking Java Programs website.

2.8 Batch Refactoring for Warnings and Migrations

Problem

Your project generates a lot of compiler/IDE warning messages. Or, you want to migrate to a new version of Java or make other sweeping changes across a codebase.

Solution

Fix the warnings now, using the warnings as guidelines, before they get out of hand. For all of the problems just mentioned, consider using a batch refactoring tool.

Discussion

As a rule of thumb, I used to predict that, once a project achieves the dubious distinction of having 38 (or perhaps 42) warning messages from Eclipse, that project would never be clean again. (In IntelliJ, Code→Inspect Code…→Analyze on a large project will usually generate hundreds or thousands of warnings, which can be discouraging.) However, "A New Hope" has arisen. While modern IDEs tend to have good refactoring menus, these require interaction on each refactoring, which takes time. There are now some automated refactoring tools, which apply refactorings without the developer needing to interact with each warning-generating line of code. Of course, the developer needs to compare the modifications with the original (e.g., `git diff`), and run all the unit tests, before committing the change to the source repository. But such tools can be a big win in terms of time. And they can help both with maintaining clean code and in making the all-through-the-code changes in migrating from use of one API to another (e.g., TestNG to JUnit).

One such tool is OpenRewrite. OpenRewrite is unsurprisingly open source, free to use forever. And it isn't limited to Java; it supports several other languages as well, though those don't concern us.

While the software could in theory be run manually, it's much easier to run using either Maven (Recipe 2.4) or Gradle (Recipe 2.5). I'm a Maven user, so I ran OpenRewrite via Maven. You can embed OpenRewrite into your *pom.xml* or *build.gradle* file, should you intend to run certain refactorings continually. For one-time use, you can just tell Maven from the command line to run the plug-in. I ran one composite recipe—`UpgradeToJava21`—on the *javasrc* source archive. Here's how that went:

```
$ mvn -U org.openrewrite.maven:rewrite-maven-plugin:run \
        -Drewrite.recipeArtifactCoordinates=org.openrewrite.recipe:rewrite-
migrate-java:RELEASE\
        -Drewrite.activeRecipes=org.openrewrite.java.migrate.UpgradeToJava21

... lots of verbiage omitted ...
```

```
[INFO] Running recipe(s)...
[WARNING] Changes have been made to main/pom.xml by:
[WARNING]     org.openrewrite.java.migrate.UpgradeToJava21
[WARNING]         org.openrewrite.java.migrate.UpgradeToJava17
[WARNING]             org.openrewrite.java.migrate.Java8toJava11
[WARNING]                 org.openrewrite.java.migrate.javax.AddInjectDependen-
cies
[WARNING]                     org.openrewrite.java.dependencies.AddDependency:
{groupId=jakarta.inject, artifactId=jakarta.inject-api, version=1.0.3, onlyIfUs-
ing=javax.inject.*, acceptTransitive=true}
[WARNING] Changes have been made to main/src/main/java/ai/ChatGptApiDemo.java
by:
[WARNING]     org.openrewrite.java.migrate.UpgradeToJava21
[WARNING]         org.openrewrite.java.migrate.UpgradeToJava17
[WARNING]             org.openrewrite.java.migrate.lang.StringFormatted
[WARNING]         org.openrewrite.java.migrate.util.SequencedCollection
[WARNING]             org.openrewrite.java.migrate.util.ListFirstAndLast
[WARNING] Changes have been made to main/src/main/java/database/NeverConcaten-
ate.java by:
[WARNING]     org.openrewrite.java.migrate.UpgradeToJava21
[WARNING]         org.openrewrite.java.migrate.UpgradeToJava17
[WARNING]             org.openrewrite.java.migrate.lang.StringFormatted

... lots more like that ...

[WARNING] Please review and commit the results.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:47 min
[INFO] Finished at: 2024-04-14T10:41:19-04:00
[INFO] ------------------------------------------------------------------------
$
```

In looking at the results, there are about 70 changed files. I tried compiling everything first and found that, while fixing at least 70 issues, it had introduced one, in fixing deprecation warnings! Here are the details:

```
  public static String converse(
     String host, int port, String path, String postBody) throws IOException {
-       URL url = new URL("http", host, port, path);
+       URL url = new URI("http", null, host, port, path, null, null).toURL();
        return converse(url, postBody);
  }
```

While replacing the deprecated `URL` constructor with the `URI` constructor, OpenRewrite failed to add a `throws` for `URISyntaxException`, which, unlike `MalformedURLException`, is not a subtype of `IOException`. That was the fix that launched a thousand source file compiles. Perhaps out of caution, the refactoring added `jakarta.inject` to the POM file, which isn't needed for anything. Three additional files still give deprecation warnings, but that's a small enough number to fix by hand!

I then went through `git diff` to make sure there weren't any unwanted changes, ran tests, and eventually committed most of it.

I did choose the "update" recipe for this demo because it doesn't require any further configuration. Obviously some of the recipes do need configuration, which you provide in a file named *recipe*. The OpenRewrite website hosts a full list of Java-specific recipes along with detailed documentation on each. I recommend this tool for those working on projects of any size.

OpenRewrite is funded by Moderne, a company and a platform to run OpenRewrite recipes at scale, across multiple codebases in multiple repositories. Quoting its website, "It's a place where platform teams and developers can collaborate to drive migrations across their codebases, mass-commit code fixes, and perform large-scale impact analyses." If you need to improve code at scale—across hundreds or thousands of repositories—check it out!

2.9 Maintaining Code Correctness with Unit Testing: JUnit

Problem

You want to have tooling to track down errors in your code.

Solution

Use unit testing to validate each class as you develop it.

Discussion

Stopping to use a debugger is time-consuming, and finding a bug in released code is much worse! It's better to *test* beforehand. The methodology of unit testing has been around for a long time; it is a tried-and-true means of getting your code tested in small blocks. Typically, in an object-oriented (OO) language like Java, unit testing is applied to individual classes, in contrast to system or integration testing where a complete slice or even the entire application is tested.

I have long been an advocate of this basic testing methodology. Indeed, developers of the software methodology known as Extreme Programming (XP for short) advocate *Test-Driven Development* (TDD): writing the unit tests *before* you write the code. They also advocate running your tests almost every time you build your application. And they ask one good question: *If you don't have a test, how do you know your code (still) works?* This group of unit-testing advocates has some well-known thought leaders, including Erich Gamma of *Design Patterns* fame and Kent Beck of *eXtreme Programming* fame (both Addison-Wesley). I definitely go along with their advocacy of unit testing.

Indeed, many of my Java source files used to come with a "built-in" test. Classes that are not main programs in their own right would often include a `main()` method that just tests out or at least exercises the functionality of the class. What surprised me is that, before encountering XP, I used to think I did this often, but an actual inspection of two projects indicated that only about a third of my classes had test cases, either internally or externally. Plus, there was no good way to find and run these. What is needed is a uniform methodology. That is provided by JUnit.

JUnit is a Java-centric methodology for providing test cases. It is hosted at *junit.org* and normally downloaded using Maven or Gradle. It is a very simple but useful testing tool. It is easy to use—you just write a test class that has a series of methods and annotate them with `@Test`. JUnit uses introspection (see Chapter 17) to find all these methods and then runs them for you. Extensions to JUnit handle tasks as diverse as load testing and testing enterprise components; the JUnit website provides links to these extensions. All modern IDEs provide built-in support for generating and running JUnit tests.

How do you get started using JUnit? All that's necessary is to write a test. Here I have written a simple test of my `Person` class and placed it into a class called `PersonTest` (note the obvious naming pattern):

```
public class PersonTest {

  Person p;

  @Test
  public void testNameConcat() {
    String act = p.getFullName();
    assertEquals(act, "Ian Darwin", "Name concatenation");
  }

  @BeforeEach
  void setup() {
    p = new Person("Ian", "Darwin");
  }
}
```

JUnit 4 has been around for ages and works well, but is being replaced by its younger sibling. JUnit 5, also known as JUnit Jupiter (after the fifth planet), is a decade or so old and has some improvements over JUnit 4. A simple test like this `PersonTest` class will be the same in JUnit 4 or 5 (but with different imports and a different before-type annotation; the one shown is from JUnit 5). Using additional features, like setup methods to be run before each test, requires different annotations between JUnit 4 and 5. OpenRewrite (see Recipe 2.8) provides recipes for migrating from JUnit 4 to 5.

You could run `PersonTest` manually, compiling the test and invoking the command-line test harness. First, download the file *junit-platform-console-standalone-1.10.3.jar* (or a later version) from the JUnit User Guide, where you can also find directions on running it. In practice, most developers find that running tests this way is incredibly tedious, due to having to set several JARs in `CLASSPATH`. Instead, just put the tests in the standard directory structure (i.e., *src/test/java/*) with the same package as the code being tested. Then run Maven (see Recipe 2.4), which will automatically compile and run all the unit tests and will halt the build if any test fails, *every time you try to build, package, or deploy your application*. Gradle will do so too.

All modern IDEs provide built-in support for running JUnit tests; in Eclipse, you can right-click a project in the Package Explorer and select Run As→Unit Test to have it find and run all the JUnit tests in the entire project. The `MoreUnit` plug-in (free in the Eclipse Marketplace) aims to simplify the creation and running of tests. IntelliJ also finds and runs tests; the `PersonTest` is shown in Figure 2-2. The checkmarks in the lower left (green in the PDF, gray in the printed book) show that all tests passed.

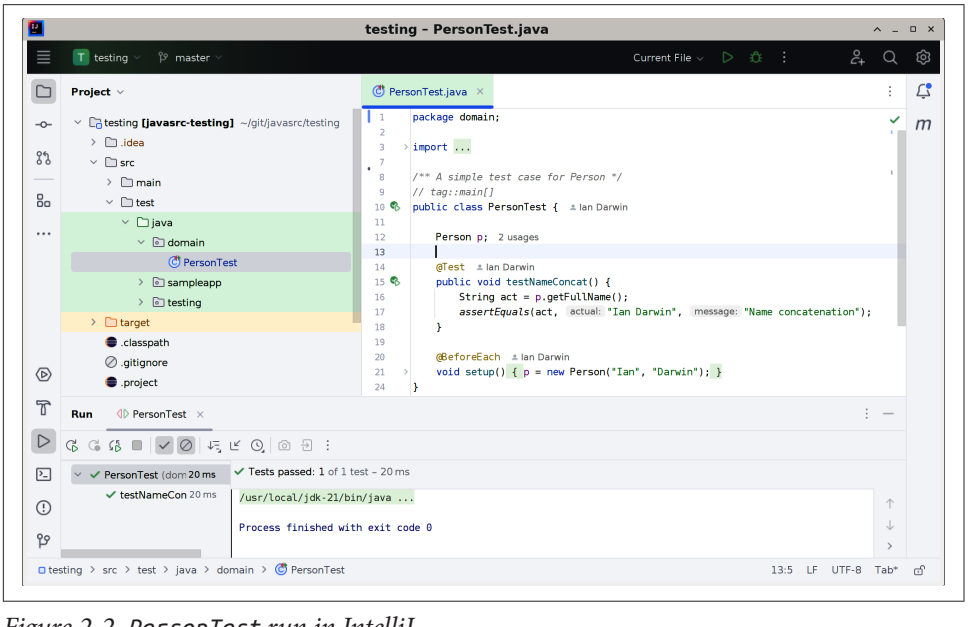


Figure 2-2. `PersonTest` run in IntelliJ

The *Hamcrest matchers* allow you to write more expressive tests at the cost of an additional download. Support for them is built into JUnit 5 via the `assertThat` static method; you just have to add them to your build file.

Here's an example of using the Hamcrest matchers:

```
/// This is really a Demo, not a Test...
public class HamcrestTest {

  @Test
  public void showGroupAssertions() {
    assertAll("Assert tests in a group",
    //  () -> assertEquals(2,3), // Sure to fail first: see if others run.
    //  () -> assertEquals(3,1+1), // Will also run if uncommented, and fail
      () -> assertEquals(2,1+1),
      () -> assertEquals("Hi", "Highlighter".substring(0,2))
    );
  }
}
```

See Also

JUnit offers considerable documentation of its own; download it from the website listed earlier.

The very ancient JUnit 3.x series predates Java 5 annotations; it used a naming convention to find and run tests. If you find any such tests, you should probably update them. OpenRewrite (Recipe 2.8) has a series of recipes for improving tests, including converting JUnit 3.x and JUnit 4 to JUnit 5. I ran this on the 65 or so tests in the *javasrc* repository; it changed all but a few of them, and left four or five (mostly JUnit 3.8 tests) needing a bit of editing before they would compile and run. Still, it saved a lot of work! In case you need it, here's the incantation I used:

```
$ mvn -U org.openrewrite.maven:rewrite-maven-plugin:run \
-Drewrite.recipeArtifactCoordinates=org.openrewrite.recipe:rewrite-testing-
frameworks:RELEASE \
-Drewrite.activeRecipes=org.openrewrite.java.testing.junit5.JUnit5BestPractices
```

An alternative unit test framework for Java is *TestNG*. It got some early traction by adopting features such as Java annotations before JUnit did; but since JUnit got with the annotations program, JUnit remained the dominant package for Java unit testing.

Another package of interest is AssertJ, which appears to offer similar power to the combination of JUnit with Hamcrest.

Don't confuse any of these with the `assert` keyword, which is placed in production code rather than in test code, and is only active when the `-ea` (enable assertions) command-line option is present at runtime.

Finally, one often needs to create substitute objects for use by the class being tested (the dependencies of the class under test). While you can code these by hand, in general I encourage use of packages such as Mockito which can generate *mock objects*

dynamically, have these mocks provide fixed return values, verify that the dependencies were called correctly, and so on. See Recipe 2.10 for details on Mockito.

Remember: *test early and often!*

2.10 Isolating the Test Target with Mock Objects and Mockito

Problem

When unit testing, it's desirable to avoid depending on the behavior of dependent objects used by the object you're testing.

Solution

Mock objects allow you to provide a fully controlled, fake version of a dependent object for your test subject to use, either before the dependency is written or to prevent the correctness of your test from depending on a class other than the one being tested.

Discussion

It often happens that you need to test an object that has dependencies, either before the dependency class is written or so the correctness of your test is independent of code outside the code you're testing. That's where mock objects come in. A mock object pretends to be the dependency but is really a replacement for it, concocted just for the purpose of testing.

The simplest form is the hardcoded mock; you simply write a class that implements the interface of (or possibly extends) the dependency:

```
public class MockWeatherStation implements WeatherStation {
        // implement required methods...
}
```

This is good enough to satisfy simple compilation, until you start to call methods. Then you can add the first few methods. Or have your IDE implement all the missing methods from the interface, but they will all return the default values of 0, `false`, or `null`. A more flexible method is to create a "dummy" object via the Mockito framework. This will conveniently create a mock object that either implements or extends the interface or class you provide it with. The following snippets are from *testing/src/test/java/sampleapp/WeatherInfoSupplierTest.java*:

```
import static org.mockito.*;
...
WeatherStation mockStation = mock(WeatherStation);
```

The methods will all still return the default values. But now you can have them return particular values, using Mockito's `when` function. You can also use a Mockito mock as a "mock with expectations," that is, expecting that a given method on the mock was called a certain number of times (or was never called), or that it was called with certain arguments, etc.:

```
when(mockStation.getTemperature()).thenReturn(19.0);
if (testCount == 0) { // hard-coded mock, no expectations
        return;
}
assertEquals(19.0, reporter.getTemperature());
// Above call is expected to delegate to station.getTemperature()
verify(mockStation, atLeastOnce()).getTemperature();
```

The `when` call specifies that when `mockStation.getTemperature()` is called (it is called by the reporter's `getTemperature()`, by simple delegation), it will return the value 19 (degrees C). This is much easier to expect in a test than whatever today's temperature is. The issue is that you can only get the current exact temperature from the actual sensor you're testing, so that would not be much of a test. Then we call the actual work method (in this case, `reporter.getTemperature()`) on the class we're testing (remember that the mock is just a bit player in this drama). Finally, we tell the mock to verify that our expectations were met regarding how the dependency was used; if not, the test will fail.

There is of course far more to Mockito. You can create mocks just by annotating fields. You can create expectations that a certain value of a certain type was passed into a method. For this and more, see the Mockito documentation.

There are other mocking frameworks, which can be found on the web.

2.11 Logging: Network or Local

Problem

Your class is running in such a way that viewing its debugging output is difficult.

Solution

Use a network-aware logger like the Java Logging API (JUL), the Apache Logging Services Project's Log4j, the open source Simple Logging Facade for Java (SLF4J), or another logger discussed here.

Discussion

Getting the debug output from a desktop client is fairly easy on most operating systems. But if the program you want to debug is running in a container like a servlet engine or an EJB server, it can be difficult to obtain debugging output, particularly if the container is running on a remote computer. It would be convenient if you could have your program send messages back to a program on your desktop machine for immediate display. Needless to say, it's not that hard to do this with Java's socket mechanism.

Many logging APIs can handle this:

- For years, Java has had a standard logging API, Java Util Logging (JUL, discussed in Recipe 2.14), that talks to various logging mechanisms, including Unix `syslog`.
- The Apache Logging Services Project produces Log4j, which is used in many open source projects that require logging (see Recipe 2.13).
- The Apache Commons Logging is a facade that lets you choose an implementation (such as JUL or Log4j). It is not discussed here.
- SLF4J (see Recipe 2.12) is the newest logging API and, as the name implies, is a facade that can use the others.
- Before these became widely used, I wrote a small, simple API to handle this type of logging function. My `netlog` is not discussed here because it is preferable to use one of the standard logging mechanisms; its code is in the *logging* subdirectory of the *javasrc* repo if you want to exhume it.

The JDK logging API, Log4j, and SFL4J are more fully fleshed out and can write to such destinations as a file; an `OutputStream` or `Writer`; or a remote Log4j, Unix `syslog`, or Windows Event Log server.

The program being debugged is the client from the logging API's point of view—even though it may be running in a server-side container such as a web server or application server—because the network client is the program that initiates the connection. The program that runs on your desktop machine is the "server" program for sockets because it waits for a connection to come along.

If you want to run any network-based logger reachable from any public network, you need to be more aware of security issues. One common form of attack is a simple denial-of-service (DoS), during which the attacker makes a lot of connections to your server in order to slow it down. If you are writing the log to disk, for example, the attacker could fill up your disk by sending lots of garbage. In common use, your log listener would be behind a firewall and not reachable from outside; but if this is not the case, beware of the DoS attack.

Some of the next few recipes include options for networking. If you're short on the details of how networking is done in Java, you might want to refer to Chapter 14. And, if what you want is to log the network traffic itself (SSL only), you can use `System.setProperty("javax.net.debug", "all")` or set this on the Java command line with `-Djava.net.debug=all`. Beware that the output will be hundreds of lines long just for opening a connection! The output can be decoded using these instructions from Oracle.

2.12 Setting Up SLF4J

Problem

You want to use a logging API that lets you use any of the other logging APIs, for example, so that your code can be used in other projects without requiring them to switch logging APIs.

Solution

Use SLF4J: get a `Logger` from the `LoggerFactory`, and use its various methods for logging.

Discussion

Using SLF4J requires only one JAR file to compile, *slf4j-api-1.x.y.jar* (where *x* and *y* will change over time). To actually get logging output, you need to add one of several implementation JARs to your runtime `CLASSPATH`, the simplest of which is *slf4j-simple-1.x.y.jar* (where *x* and *y* should match between the two files).

Once you've added those JAR files to your build script or on your `CLASSPATH`, you can get a `Logger` by calling `LoggerFactory.getLogger()`, passing either the string name of a class or package or just the current `Class` reference. Then call the logger's logging methods. A simple example is in Example 2-4.

Example 2-4. main/src/main/java/logging/Slf4jDemo.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Slf4jDemo {

  final static Logger theLogger =
      LoggerFactory.getLogger(Slf4jDemo.class);

  public static void main(String[] args) {
```

```
    Object o = new Object();
    theLogger.info("I created this object: " + o);

  }
}
```

There are various methods used to log information at different levels of severity, which are shown in Table 2-2.

Table 2-2. SLF4J logging methods

Name	Meaning
`trace`	Verbose debugging (disabled by default)
`debug`	Verbose debugging
`info`	Low-level informational message
`warn`	Possible error
`error`	Serious error

One of the advantages of SLF4J over most of the other logging APIs is the avoidance of the dead string antipattern. When using many other logger APIs, you may find code like the following:

```
logger.log("The value is " + object + "; this is not good");
```

This can lead to a performance problem, in that the object's `toString()` is implicitly called, and two string concatenations are performed, before we even know if the logger is going to use them! In code that is called repeatedly, this can lead to a lot of overhead.

This led the other logging packages to offer code guards, based on logger methods that can find out very quickly if a logger is enabled, leading to code like the following:

```
if (logger.isEnabled()) {
        logger.log("The value is " + object + "; this is not good");
}
```

This solves the performance problem but clutters the code! SLF4J's solution is to use a mechanism similar to (but not quite compatible with) Java's `MessageFormat` mechanism, as shown in Example 2-5.

Example 2-5. main/src/main/java/logging/Slf4jDemo2.java

```
public class Slf4jDemo2 {

  final static Logger theLogger = LoggerFactory.getLogger(Slf4jDemo2.class);

  public static void main(String[] args) {
```

```
    try {
      Person p = new Person();
      // populate person's fields here...
      theLogger.info("I created an object {}", p);

      if (p != null) {  // bogus, just to show logging
        throw new IllegalArgumentException("Just testing");
      }
    } catch (Exception ex) {
      theLogger.error("Caught Exception: " + ex, ex);
    }
  }
}
```

Although this doesn't actually demonstrate network logging, it is easy to accomplish that in conjunction with a logging implementation like Log4j or JUL (a standard part of the JDK), which allows you to provide configurable logging. Log4j is described in the next recipe.

See Also

The SLF4J website contains a manual that discusses the various `CLASSPATH` options. There are also some Maven artifacts for the various options.

2.13 Network Logging with Log4j

Problem

You wish to write logfile messages using Log4j.

Solution

Get a `Logger` and use its `log()` method or the convenience methods. Control logging by changing a properties file. Use the org.apache.logging.log4j.net package to make it network based.

Discussion

Logging using Log4j is simple, convenient, and flexible. You need to get a `Logger` object from the static method `LogManager.getLogger()`. The `Logger` has public void methods (`debug()`, `info()`, `warn()`, `error()`, and `fatal()`), each of which takes one `Object` to be logged (and an optional `Throwable`). As with `System.out.println()`, if you pass in anything that is not a `String`, its `toString()` method is called. A generic logging method is also included:

```
public void log(Level level, Object message);
```

The `Level` class is defined in the Log4j 2 API. The standard levels are, in order, `DEBUG` < `INFO` < `WARN` < `ERROR` < `FATAL`. That is, debug messages are considered the least important, and fatal messages the most important. Each `Logger` has a level associated with it; messages whose level is less than the `Logger`'s level are silently discarded.

This recipe describes Version 2 of the Log4j API. Between Version 1 and Version 2, there were changes to the package names, filenames, and the method used to obtain a `Logger`. If you see code using, for example, `Logger.getLogger("class name")` rather than the newer `LogManager.getLogger()`, that code is written to the older API, which is no longer maintained (the Log4j website refers to Log4j 1.2, and versions up to 2.12, as "legacy"; we are using 2.13 in this recipe). However, a good degree of compatibility is offered for code written to the 1.x API; see the Log4J documentation.

A simple application can log messages using these few statements:

```
public class Log4JDemo {

  private static Logger myLogger = LogManager.getLogger();

  public static void main(String[] args) {

    Object o = new Object();
    myLogger.info("I created an object: " + o);

  }
}
```

If you compile and run this program with no *log4j2.properties* file, it does not produce any logging output (see the *log4j2demos* script in the source folder). We need to create a configuration file whose default name is *log4j2.properties*. You can also provide the logfile name via system properties: `-Dlog4j.configurationFile=URL`.

Log4j configuration is very flexible, and therefore very complex. Even Apache's own documentation admits that "trying to configure Log4j without understanding [the logging architecture] will lead to frustration." See the Apache website for full details on the logging configuration file location and format.

Every `Logger` has a `Level` to specify what level of messages to write. It will also have an `Appender`, which is the code that writes the messages out. A `ConsoleAppender` writes to `System.out`, of course; other loggers write to files, operating system–level loggers, and so on. A simple configuration file looks something like this:

```
# WARNING - log4j2.properties must be on your CLASSPATH,
# not necessarily in your source directory.

# The configuration file for Version 2 is different from V1!

rootLogger.level = info
rootLogger.appenderRef.stdout.ref = STDOUT

appender.console.name = STDOUT
appender.console.type = Console
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %m%n
appender.console.filter.threshold.type = ThresholdFilter
appender.console.filter.threshold.level = debug
```

This file gives the root logger a level of `DEBUG`, which causes it to write all messages. The config file also sets up an appender named `STDOUT`, which is configured on the next few lines. Note that I didn't have to refer to the `com.darwinsys Logger`. Because every `Logger` inherits from the root logger, a simple application needs to configure only the root logger. The properties file can also be an XML document, or you can write your own configuration parser (almost nobody does this).

If the logging configuration file is not found, the root logger defaults to `Level.ERROR`, so you will not see any output below the `ERROR` level.

With the configuration file in place, the demonstration works better. Running this program (with the appropriate `CLASSPATH` as done in the scripts) produces this output:

```
$ java Log4j2Demo
I created an object: java.lang.Object@477b4cdf
$
```

A common use of logging is to show a caught `Exception`, as shown in Example 2-6.

Example 2-6. main/src/main/java/Log4JDemo2.java (Log4j—catching and logging)

```
public class Log4JDemo2 {

  private static Logger myLogger = LogManager.getLogger();
```

```
  public static void main(String[] args) {

    try {
      Object o = new Object();
      myLogger.info("I created an object: " + o);
      if (o != null) {  // bogus, just to show logging
        throw new IllegalArgumentException("Just testing");
      }
    } catch (Exception ex) {
      myLogger.error("Caught Exception: " + ex, ex);
    }
  }
}
```

When run, `Log4JDemo2` produces the expected output:

```
$ java Log4JDemo2
I created an object: java.lang.Object@477b4cdf
Caught Exception: java.lang.IllegalArgumentException: Just testing
java.lang.IllegalArgumentException: Just testing
        at logging.Log4JDemo2.main(Log4JDemo2.java:17) [classes/:?]
$
```

Much of the flexibility of Log4j 2 stems from its use of external configuration files; you can enable or disable logging without recompiling the application. A properties file that eliminates most logging might have this entry:

```
rootLogger.level = fatal
```

Only fatal error messages print; all levels less than that are ignored.

To log from a client to a server on a remote machine, the `SocketAppender` can be used. There is also an `SmtpAppender` to send urgent notices via email. See the Log4j documentation for details on all the supported appenders. Here is *log4j2-network.properties*, the socket-based networking version of the configuration file:

```
# WARNING - log4j2.properties must be on your CLASSPATH,
# not necessarily in your source directory.

# The configuration file for Version 2 is different from V1!

rootLogger.level = info
rootLogger.appenderRef.stdout.ref = STDOUT

appender.console.type = Socket
appender.console.name = STDOUT
appender.console.host = localhost
appender.console.port = 6666
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %m%n
appender.console.filter.threshold.type = ThresholdFilter
appender.console.filter.threshold.level = debug
```

This file gets passed to the demo programs via a Java system property in the `netdemos` script:

```
build=../../../../target/classes
log4j2_jar=\
${HOME}/.m2/repository/org/apache/logging/log4j/log4j-api/2.13.0/log4j-
api-2.13.0.jar:\
${HOME}/.m2/repository/org/apache/logging/log4j/log4j-core/2.13.0/log4j-
core-2.13.0.jar

echo "==> Log4JDemo"
java -Dlog4j.configurationFile=log4j2-network.properties \
        -classpath ".:${build}:${log4j2_jar}" logging.Log4JDemo

echo "==> Log4JDemo2"
java -Dlog4j.configurationFile=log4j2-network.properties \
        -classpath ".:${build}:${log4j2_jar}" logging.Log4JDemo2
```

When run with the *log4j2-network.properties* file, you have to arrange for a listener on the other end. On Unix systems the `nc` (or `netcat`) program will work fine:

```
$ nc -kl 6666
I created an object: java.lang.Object@37ceb1df
I created an object: java.lang.Object@37ceb1df
Caught Exception: java.lang.IllegalArgumentException: Just testing
java.lang.IllegalArgumentException: Just testing
        at logging.Log4JDemo2.main(Log4JDemo2.java:17) [classes/:?]
^C
$
```

Netcat option `-l` says to listen on the numbered port; `-k` tells it to keep listening, that is, to reopen the connection when the client closes it, as happens when each demo program exits.

There is a performance issue with some logging calls. Consider some expensive operation, like a `toString()` or two along with several string concatenations passed to a `Log.info()` call in an often-used piece of code. If this is placed into production with a higher logging level, all the work will be done, but the resultant string will never be used. In older APIs we used to use "code guards," methods like `isLoggerEnabled(Level)`, to determine whether to bother creating the string. Nowadays, the preferred method is to create the string inside a lambda expression (see Chapter 9). All the log methods have an overload that accepts a `java.util.function.Supplier` argument (Example 2-7).

Example 2-7. main/src/main/java/logging/Log4JLambda.java

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Log4JLambda {
```

```
private static Logger myLogger = LogManager.getLogger();

public static void main(String[] args) {

  Person customer = getPerson();
  myLogger.info( () ->
    "Value %d from Customer %s".formatted(customer.value, customer) );

}
```

This way the string operations will only be performed if needed: if the `Logger` is operating at the `INFO` level it will call the `Supplier` and if not, it won't do the expensive operation.

When run as part of the *log4j2demos* script, this prints:

```
Value 42 from Customer Customer[Robin]
```

For more information on Log4j, visit its main website. Log4j 2 is free software, distributed under the Apache Software Foundation license.

2.14 Network Logging with java.util.logging

Problem

You wish to write logging messages without having to add any logging dependencies.

Solution

Using the Java logging mechanism in `java.util.logging`, get a `Logger`, and use it to log your messages and/or exceptions.

Discussion

The Java Logging API (package java.util.logging) is similar to, and was obviously inspired by, the Log4j package, which it was presumably intended to replace, or at least standardize. It's a bit less flexible than Log4j, but has the advantage that it's in the JDK, so no dependencies need to be added for it.

You acquire a `Logger` object by calling the static `Logger.getLogger()` with a descriptive `String`. You then use instance methods to write to the log; these methods include the following:

```
public void log(java.util.logging.LogRecord);
public void log(java.util.logging.Level,String);
// and a variety of overloaded log(  ) methods
public void logp(java.util.logging.Level,String,String,String);
public void logrb(java.util.logging.Level,String,String,String,String);
```

```
    // Convenience routines for tracing program flow
    public void entering(String,String);
    public void entering(String,String,Object);
    public void entering(String,String,Object[]);
    public void exiting(String,String);
    public void exiting(String,String,Object);
    public void throwing(String,String,Throwable);

    // Convenience routines for log(  ) with a given level
    public void severe(String);
    public void warning(String);
    public void info(String);
    public void config(String);
    public void fine(String);
    public void finer(String);
    public void finest(String);
```

As with Log4j, every `Logger` object has a given logging level, and messages below that level are silently discarded:

```
    public void setLevel(java.util.logging.Level);
    public java.util.logging.Level getLevel(  );
    public boolean isLoggable(java.util.logging.Level);
```

As with Log4j, objects handle the writing of the log. Each `Logger` has a `Handler`:

```
    public synchronized void addHandler(java.util.logging.Handler);
    public synchronized void removeHandler(java.util.logging.Handler);
    public synchronized java.util.logging.Handler[] getHandlers(  );
```

Each `Handler` has a `Formatter`, which formats a `LogRecord` for display. By providing your own `Formatter`, you have more control over how the information being passed into the log gets formatted.

Unlike Log4j, the Java SE logging mechanism has a default configuration, so Example 2-8 is a minimal logging example program.

Example 2-8. main/src/main/java/logging/JulLogDemo.java

```
import java.util.logging.Logger;

public class JulLogDemo {
  public static void main(String[] args) {

    Logger myLogger = Logger.getLogger("com.darwinsys");

    Object o = new Object();
    myLogger.info("I created an object: " + o);
  }
}
```

Running it prints the following:

```
$ juldemos
Jan 31, 2020 1:03:27 PM logging.JulLogDemo main
INFO: I created an object: java.lang.Object@5ca881b5
$
```

As with Log4j, one common use is in logging caught exceptions; the code for this is in Example 2-9. This starts by loading a configuration file, shown following the example.

Example 2-9. main/src/main/java/logging/JulLogDemo2.java (catching and logging an exception)

```
public class JulLogDemo2 {
  public static void main(String[] args) {

    // Must be a full path; does not use getResourceAsStream()
    final Path path = Path.of("target/classes" + "/" +
        "logging.properties");
    if (!Files.exists(path)) {
      System.out.printf("path %s non-existent\n", path);
    }
    System.setProperty("java.util.logging.config.file", path.toString());
    System.out.println(System.getProperty("java.util.logging.config.file"));

    Logger logger = Logger.getLogger("com.darwinsys");

    try {
      Object o = new Object();
      logger.info("I created an object: " + o);
      if (o != null) {  // just a demo to show exception logging
        throw new IllegalArgumentException("Just testing");
      }
    } catch (Exception t) {
      logger.log(Level.SEVERE, "Caught Exception", t);
    }
  }
}
```

Note that the `logging.properties` must be specified as a full path, so you have to know where it will be located at runtime. Maven and IntelliJ copy files from *src/main/resources* into the standard output folder, *target/classes*, so that is where we specify the path for the file. Here is the configuration file:

```
# Default handler
handlers=java.util.logging.ConsoleHandler,java.util.logging.SocketHandler

# Use a short log formatter (server apps often add timestamp &c).
java.util.logging.ConsoleHandler.formatter=logging.JULLogFormatter
java.util.logging.SocketHandler.formatter=logging.JULLogFormatter
```

```
java.util.logging.SocketHandler.host=127.0.0.1
java.util.logging.SocketHandler.port=5678

# Levels are: FINEST, FINER, FINE, CONFIG, INFO, WARNING, SEVERE
# ALL and NONE are potentially useful pseudo-levels.
# NONE can not be set for a logger level

# Default global logging level.
# Loggers and Handlers may override this level
.level=INFO

# Loggers are usually attached to packages.
# The global level is used by default, so levels
# specified here simply act as an override.
com.darwinsys.level=FINE
com.darwinsys.ui.level=ALL
com.darwinsys.model.level=CONFIG
com.darwinsys.data.level=SEVERE
com.darwinsys.jullambda.level=FINE
```

Because the code asks for the logger `com.darwinsys`, it will log with a level of `FINE`, analogous to `DEBUG` in `log4j`.

The logging output shows up on the standard output as:

```
Oct 14, 2024 7:24:16 PM logging.JulLogDemo2 main
INFO: I created an object: java.lang.Object@7506e922
Oct 14, 2024 7:24:16 PM logging.JulLogDemo2 main
SEVERE: Caught Exception
java.lang.IllegalArgumentException: Just testing
        at logging.JulLogDemo2.main(JulLogDemo2.java:28)
```

Because we also configured a socket handler, it will also show up on the TCP port, by default in (verbose!) XML. For this demo I used netcat (`nc`) listening on the port in the config file:

```
$ nc -l 5678
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2024-10-14T23:19:46.227009859Z</date>
  <sequence>0</sequence>
  <logger>com.darwinsys</logger>
  <level>INFO</level>
  <class>logging.JulLogDemo2</class>
  <method>main</method>
  <message>I created an object: java.lang.Object@7506e922</message>
</record>
...
```

As with Log4j, `java.util.logging` accepts a lambda expression (and has since Java 8); see Example 2-10.

Example 2-10. main/src/main/java/logging/JulLambdaDemo.java

```
/** Demonstrate how Java 8 Lambdas avoid extraneous object creation
 * @author Ian Darwin
 */
public class JulLambdaDemo {
  public static void main(String[] args) {

    Logger myLogger = Logger.getLogger("com.darwinsys.jullambda");

    Object o = new Helper();

    // If you change the log call from finest to info,
    // you see both the systrace from the toString,
    // and the logging output. As it is here,
    // you don't see either, so the toString() is not called!
    myLogger.finest(() -> "I created this object: " + o);
  }

  static class Helper {
    public String toString() {
      System.out.println("JulLambdaDemo.Helper.toString()");
      return "failure!";
    }
  }
}
```

2.15 Maintaining Your Code with Continuous Integration

Problem

You want to be sure that your entire codebase compiles and passes its tests periodically.

Solution

Use a continuous integration (CI) server such as Jenkins or TeamCity.

Discussion

If you haven't previously used CI, you are going to wonder how you got along without it. CI is simply the practice of having all developers on a project periodically *integrate* (e.g., commit) their changes into a single master copy of the project's source and then build and test the project to make sure it still works and passes its tests. This might be a few times a day, or every few days, but should not be more than that or else the integration will likely run into larger hurdles where multiple developers have modified the same file.

But it's not just big projects or program-code projects that benefit from CI. Even on a one-person project, it's great to have a single button you can click that will check out the latest version of everything, compile it, link or package it, run all the automated tests, and give a red or green pass/fail indicator. Better yet, it can do this automatically every day or even on every commit to the master branch. If you have a number of small websites, putting them all under CI control is one of several ways of moving toward an automated, DevOps culture around website deployment and management.

If you are new to the idea of CI, I can do no better than to plead with you to read Martin Fowler's insightful (as ever) paper on the topic. One of the key points is to automate both the *management* of the code *and* all the other artifacts needed to build your project, and to automate the actual process of *building* it, possibly using one of the build tools discussed earlier in this chapter.[1]

There are many CI servers, both free and commercial. In the open source world, GitHub Actions, Jenkins, and CruiseControl are some of the better known CI servers that you deploy yourself. There are also commercial/hosted solutions such as Travis CI, TeamCity, or CircleCI. TeamCity is from the developers of IntelliJ IDEA, so it probably has the best integration with the IntelliJ IDEs. These hosted servers eliminate the need for setting up and running your own CI server. They also tend to have their configuration right in your repo (*travis.yml*, etc.) so deployment to them is simplified.

Jenkins runs as a web application, either inside a Jakarta EE server or in its own standalone web server. Once it's started, you can use any standard web browser as its user interface. Installing and starting Jenkins can be as simple as unpacking a distribution and invoking it as follows:

```
java -jar jenkins.war
```

This will start up its own tiny web server. If you do that, *be sure to configure security* if your machine is reachable from the internet!

Many people find it more secure to run Jenkins in a full-function Java EE or Java web server; anything from Tomcat to JBoss to WebSphere or WebLogic will do the job and let you impose additional security constraints.

Once Jenkins is up and running and you have enabled security and are logged in on an account with sufficient privilege, you can create *jobs*. A job usually corresponds to one project, both in terms of origin (one source code checkout) and in terms of results (one *.war* file, one executable, one library, one whatever). Setting up a project

1 If the deployment or build includes a step like "Get Smith to process file X on his desktop and copy to the server," you probably don't quite get the notion of automated testing.

is as simple as clicking the New Job button at the top left of the dashboard, as shown in Figure 2-3.

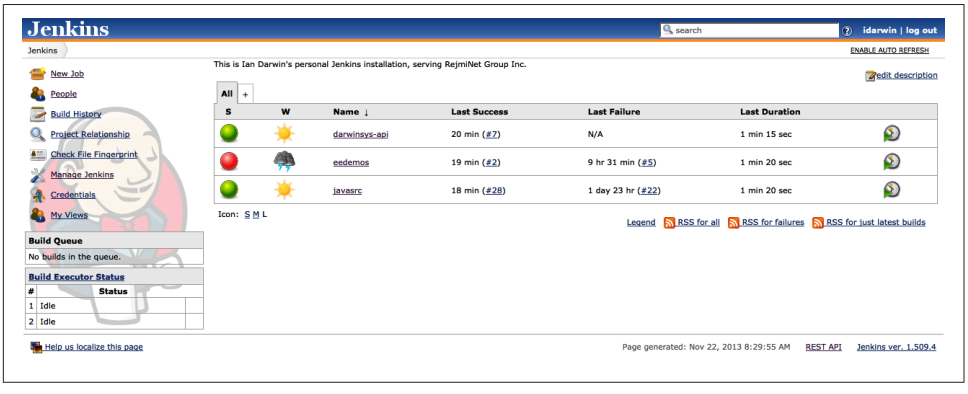


Figure 2-3. The dashboard in Jenkins

Figure 2-4 shows the first few steps of setting up a new job. You can fill in the first few pieces of information: the project's name and a brief description. Note that each and every input field has a question mark icon beside it, which will give you hints as you go along. Don't be afraid to peek at these hints!

In the next few sections of the form, Jenkins uses dynamic HTML to make entry fields appear based on what you've checked. My demo project "TooSmallToFail" starts off with no Source Code Management (SCM) repository, but your real project is probably already in Git, Subversion, or some other SCM. Don't worry if yours is not listed; there are hundreds of plug-ins to handle almost any SCM. Once you've chosen your SCM, you will enter the parameters to fetch the project's source from that SCM repository, using text fields that ask for the specifics needed for that SCM: a URL for Git, a CVSROOT for CVS, and so on.

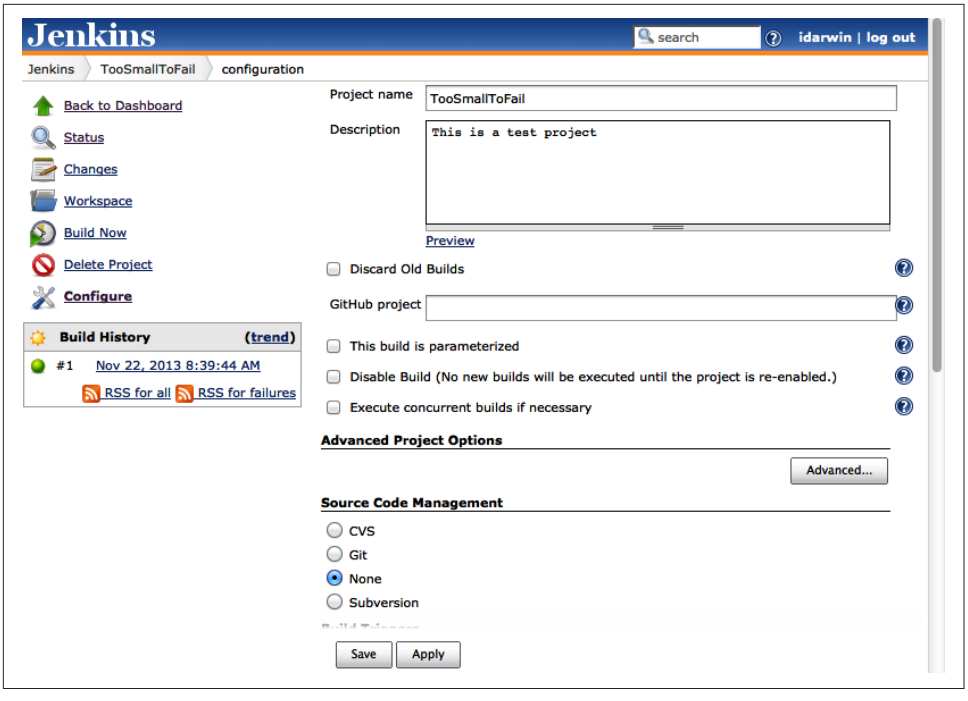


Figure 2-4. Creating a new job in Jenkins

You also have to tell Jenkins *when* and *how* to build (and package, test, deploy…) your project. For the *when*, you have several choices such as building it after another Jenkins project, building it every so often based on a cron-like schedule, or based on polling the SCM to see if anything has changed (using the same cron-like scheduler). If your project is at GitHub (not just a local Git server), or some other SCMs, you can have the project built whenever somebody pushes changes up to the repository. It's all a matter of finding the right plug-ins and following the documentation for them.

Then we have the *how*, or the build process. Again, a few build types are included with Jenkins, and many more are available as plug-ins: I've used Apache Maven, Gradle, the traditional Unix `make` tool, and even shell or command lines. As before, text fields specific to your chosen tool will appear once you select the tool. In the toy example, `TooSmallToFail`, I just use the shell command `/bin/false` (which should be present on any Unix or Linux system) to ensure that the project does, in fact, fail to build, just so you can see what that looks like.

You can have zero or more build steps; just keep clicking the Add button and add additional ones, as shown in Figure 2-5.

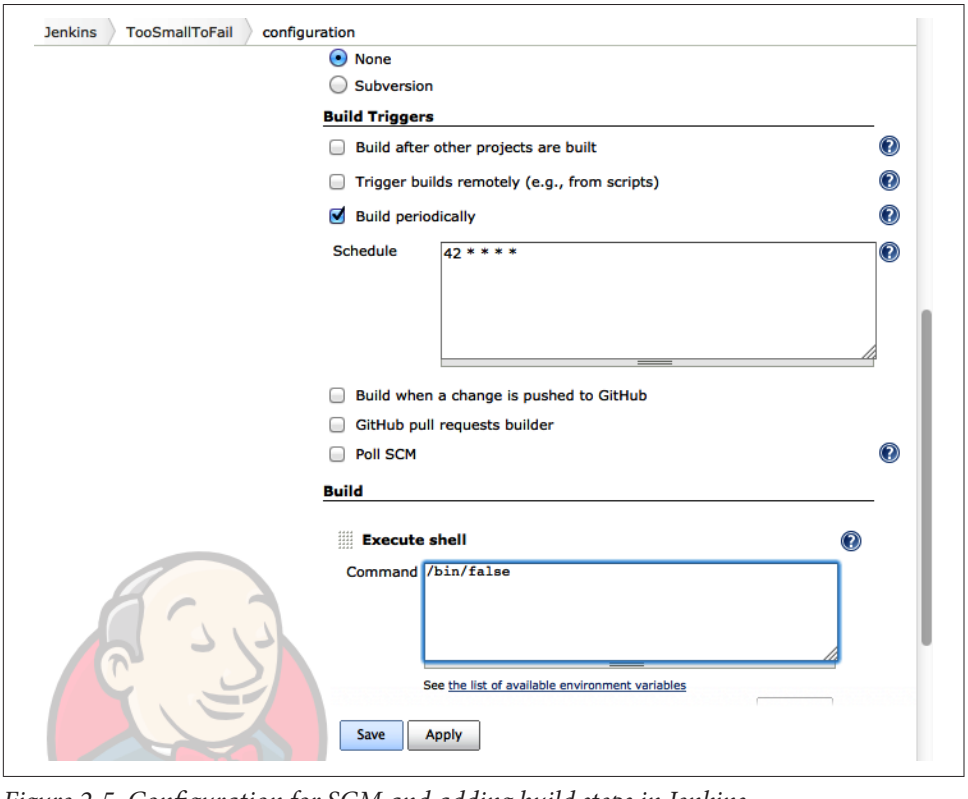


Figure 2-5. Configuration for SCM and adding build steps in Jenkins

Once you think you've entered all the necessary information, click the Save button at the bottom of the page, and you'll go back to the project's main page. Here you can click the funny little Build Now icon at the far left to initiate a build right away. Or if you have set up build triggers, you could wait until they kick in; but then again, wouldn't you rather know right away whether you've got it just right? Figure 2-6 shows the build starting.

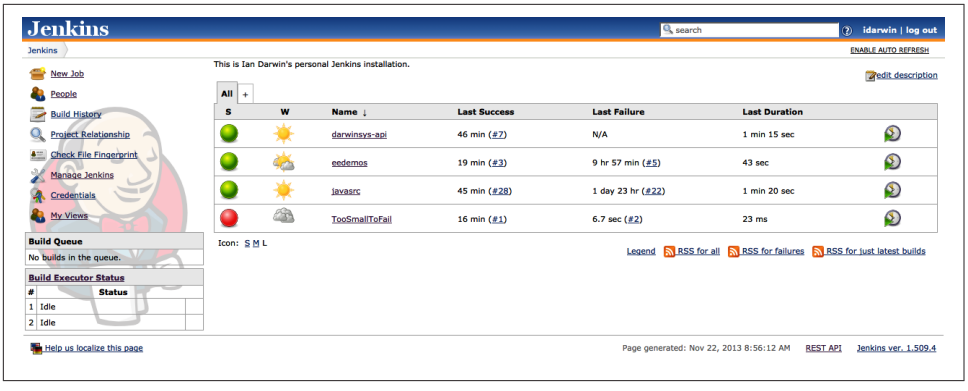


Figure 2-6. After a new job is added in Jenkins

Should a job fail to build, you get a red ball instead of a green one. Actually, a successful build shows a blue ball by default (the *go* bulb in many traffic lights in Japan, where Hudson/Jenkins's original author Kohsuke Kawaguch lives, is blue rather than green), but most people outside Japan prefer green for success, so the optional Green Balls plug-in is often one of the first to be added to a new installation.

Beside the red or green ball, you will see a "weather report" ranging from sunny (the last several builds have succeeded) to cloudy, rainy, or stormy (no recent builds have succeeded).

Click the link to the project that failed, and then the link to Console Output, and figure out what went wrong. The usual workflow is then to make changes to the project, commit/push them to the source code repository, and that may trigger the Jenkins build again.

There are *hundreds* of optional plug-ins for Jenkins. To make your life easier, almost all of them can be installed by clicking the Manage Jenkins link and then going to Manage Plug-ins. The Available tab lists all the plug-ins that are available from *Jenkins.io*; you just need to click the checkbox beside the ones you want, and click Apply. You can also find updates there. If your plug-in addition or upgrade requires a restart, you'll see a yellow ball and words to that effect; otherwise you should see a green (or blue) ball indicating plug-in success. You can also see the list of plug-ins directly on the web.

If you use a different CI system, you'll need to check that system's documentation, but the concepts and the benefits will be similar.

2.16 Performance Timing

Problem

Slow performance?

Solution

Use a *profiler*, or time individual methods using `System.currentTimeMillis()` before and after invoking the target method; the difference is the time that method took.

Discussion

Profilers

Profiling tools—profilers—have a long history as one of the important tools in a programmer's toolkit. A commercial profiling tool will help find bottlenecks in your program by showing both the number of times each method was called and the amount of time in each.

Quite a bit of useful information can be obtained from a Java application by use of the `VisualVM` tool, which was part of the Oracle JDK up until Java 8. With Java 9 this tool was released as open source, and it's now available from the VisualVM project.

Another tool that is part of the JDK is Java Flight Recorder, which is now open source and built into the JDK. Its data is meant to be analyzed by Java Mission Control. There are also third-party profilers that will give more detailed information; a web search will find current commercial offerings.

Measuring a single method

The simplest technique is to save the JVM's accumulated time before and after dynamically loading a main program and then calculate the difference between those times. Code to do just this, using the Reflection API, is presented in Example 17-26; for now, just remember that we have a way of timing a given Java class.

One way of measuring the efficiency of a particular operation is to run it many times in isolation. The overall time the program takes to run thus approximates the total time of many invocations of the same operation. Gross numbers like this can be compared if you want to know which of two ways of doing something is more efficient. Consider the case of string concatenation versus `println()`. The code:

```
println("Time is " + n.toString( ) + " seconds");
```

will probably work by creating a `StringBuilder`; appending the string `"Time is"`, the value of `n` as a string, and `"seconds"`; and finally converting the finished `StringBuilder` to a `String` and passing that to `println()`. Suppose you have a program that does a lot of this, such as a Java servlet that creates a lot of HTML this way, and you expect (or at least hope) your website to be sufficiently busy so that doing this efficiently will make a difference. There are two ways of thinking about this:

- Theory A: this string concatenation is inefficient.
- Theory B: string concatenation doesn't matter; `println()` is inefficient, too.

A proponent of Theory A might say that because `println()` just puts stuff into a buffer, it is very fast and that string concatenation is the expensive part.

How to decide between Theory A and Theory B? Assume you are willing to write a simple test program that tests both theories. Let's just write a simple program both ways and time it. Example 2-11 is the timing program for Theory A.

Example 2-11. main/src/main/java/performance/StringPrintA.java

```
public class StringPrintA {
  public static void main(String[] argv) {
    Object o = "Hello World";
    for (int i=0; i<100000; i++) {
      System.out.println("<p><b>" + o.toString() + "</b></p>");
    }
  }
}
```

`StringPrintAA` (in the *javasrc* repo but not printed here) is the same but explicitly uses a `StringBuilder` for the string concatenation. Example 2-12 is the tester for Theory B.

Example 2-12. main/src/main/java/performance/StringPrintB.java

```
public class StringPrintB {
  public static void main(String[] argv) {
    Object o = "Hello World";
    for (int i=0; i<100000; i++) {
      System.out.print("<p><b>");
      System.out.print(o.toString());
      System.out.print("</b></p>");
      System.out.println();
    }
  }
}
```

Timing results

I ran `StringPrintA`, `StringPrintAA`, and `StringPrintB` twice each on the same computer. To eliminate JVM startup times, I ran them from a program called `TimeNoArgs`, which takes a class name and invokes its `main()` method, using the Reflection API. `TimeNoArgs` and a shell script to run it, *stringprinttimer.sh*, are in the *performance* folder of the *javasrc* source repository. Here are the results:

2004 program	Seconds
`StringPrintA`	17.23, 17.20 seconds
`StringPrintAA`	17.23, 17.23 seconds
`StringPrintB`	27.59, 27.60 seconds

2014 program	Seconds
`StringPrintA`	0.714, 0.525 seconds
`StringPrintAA`	0.616, 0.561 seconds
`StringPrintB`	1.091, 1.039 seconds

2024 program	Seconds
`StringPrintA`	0.146, 0.075 seconds
`StringPrintAA`	0.098, 0.08 seconds
`StringPrintB`	0.298, 0.282 seconds

Although the times went down by around two orders of magnitude over two decades due to JVM improvements and faster hardware (Moore's Law), the ratios remain consistent: `StringPrintB`, which calls `print()` and `println()` multiple times, takes very roughly twice as long.

Moral: don't guess. If it matters, time it.

Another moral: multiple calls to `System.out.print()` cost more than the same number of calls to a `StringBuilder`'s `append()` method, by a factor of roughly 1.5 (or 150%). Theory B wins; the extra `println` calls appear to save a string concatenation but make the program take substantially longer.

Other aspects of performance: Garbage collection

There are many other aspects of software performance. One that is fundamental to Java is garbage collection (GC) behavior. Sun/Oracle usually discusses this at JavaOne. For example, see the 2003 JavaOne presentation "Garbage Collection in the Java HotSpot Virtual Machine". See also the 2007 JavaOne talk by the same GC development team, "Garbage Collection-Friendly Programming," TS-2906.

JavaOne 2010 featured an updated presentation entitled "The Garbage Collection MythBusters".

2.17 Creating a Custom JDK Distribution with jlink

Problem

You are distributing your application to end users, and you want to minimize the size of your download.

Solution

Modularize your application (see Recipe 2.3), use `jdeps` to get a complete list of the modules it uses, then use `jlink` to create the mini-Java, and distribute that to your users.

Discussion

`jlink` is a command-line tool introduced in Java 9 that can make up a mini-Java distribution containing only your application and the JDK classes it uses. That is, it omits any of the thousands of JDK classes that your app will never use.

First, you need to compile and package your `module-info` and your application code. You can use Maven or Gradle, or just use the JDK tools directly:

```
$ javac -d . src/*.java
$ jar cvf demo.jar module-info.class  demo
```

If you wish to see the list of modules that will be included, you can optionally run the `jdeps` tool to get this list:

```
$ jdeps --module-path . demo.jar
demo
 [file:///Users/ian/workspace/javasrc/jlink/./]
   requires mandated java.base (@11.0.2)
demo -> java.base
   demo                  -> java.io            java.base
   demo                  -> java.lang          java.base
```

Once the classes have been compiled, you can run the `jlink` tool to build a mini-Java distribution with your demo app embedded:

```
jlink --module-path . --no-header-files \
    --no-man-pages --compress=2 --strip-debug \
    --launcher rundemo=demo/demo.Hello \
    --add-modules demo --output mini-java
```

The `--launcher` *`name=module/main`* argument asks `jlink` to create a script file named *name* to run the application.

Not all options will be needed. In one application of mine, I did the following, which also shows the disk savings by not including the entire JDK:

```
$ jlink --output myapp-runtime --add-modules java.base,java.desktop,java.logging
$ du -sh myapp-runtime/
252M    myapp-runtime/
$ du -sh /usr/local/jdk-21
381M    /usr/local/jdk-21
$
```

There was a 33% reduction in the size of the regular JDK after creating the mini-JDK.

If you got no errors, you should be able to run the mini-JDK either with the `java` command or with the generated shell script:

```
$ mini-java/bin/java demo.Hello
Hello, world.
$ mini-java/bin/rundemo
Hello, world.
$
```

You might want to copy the entire mini-Java folder to a machine that doesn't have a regular Java installation and run it there in order to be sure you don't have any missing dependencies.

The concept of a mini-distribution is appealing, but you must consider these issues:

- Any non-modularized libraries will cause failure.
- There is no upgrade mechanism for such mini-Javas. These are quite suitable for microservice deployments where you rebuild often. For applications shipped to customers, though, you'd have to regenerate them and get your customers to download and reinstall (on short notice whenever there's a security update).
- Disk space is generally no longer expensive relative to the cost of your time in maintaining such a distribution.

Thus, you have to decide if this is worthwhile for your application.

2.18 Creating Platform-Specific Installers with jpackage

Problem

14 You have a desktop or command-line application that users will download and install on their systems. You want your users to be able to install your app using platform-appropriate installers such as `rpm`/`deb` on Linux, `dmg` on macOS, and `MSI` on Microsoft Windows, and without their having to download specific versions of Java.

Solution

Use `jpackage`, which creates platform-specific installers that include a customized JDK that will include the APIs needed to run your application.

Discussion

Java 14 introduced two tools, `jlink` and `jpackage` (they were in preview before this). The `jpackage` command can create *.msi* or *.exe* installers for Microsoft Windows, *pkg* or *dmg* files for macOS, and *app-image*, *rpm*, or *deb* files for Linux. These installers include your Java *.class* files, JAR file dependencies, and the parts of the JDK needed to run your app. This saves users from having to download a JDK, keep track of different versions of Java for different apps, etc. It trades off a bit of disk space to get rid of a lot of maintenance and incompatibility issues.

Unsurprisingly, `jpackage` must be run on the platform for which you want to make an installer, because it's going to copy parts of the running JDK into the installer. And because it works on several operating systems, it has quite a few command-line arguments, some general and some platform-specific. A complete list would take too much space here, but it can be found in the JDK documentation.

On Microsoft Windows you need to have installed an installer maker, which you will be prompted to install the first time you run `jpackage` on that OS.

The following is the `mkinstaller` script I use to prepare installers for `pdfshow`, a desktop app that I wrote and maintain. The complete version of the script is in this GitHub repository, and the same script gets run on all three major desktop platforms. If you want to adopt my script on Microsoft Windows, you'd need a Unix-like shell such as the `git bash` download. This will be less work than rewriting it into a Windows-only PowerShell script. I've omitted the part of the script that sets numerous *shell variables* such as the installer format (`fmt`) and icon format (`icon_format`) to use based on the operating system, and some platform-specific options (`OS_SPECIFIC`). The key part of the script is shown here:

```
jpackage \
--name PDFShow \
--app-version ${RELEASE_VERSION} \
--license-file LICENSE.txt \
--vendor "Rejminet Group Inc." \
--type "${fmt}" \
--icon src/main/resources/images/logo.${icon_format} \
--input target \
--main-jar pdfshow-${RELEASE_VERSION}-jar-with-dependencies.jar \
--runtime-image pdfshow-runtime \
${OS_SPECIFIC} || {
        echo jpackage did not complete normally!
        exit 1
}
```

The last few lines may give pause if you've not used a `*NIX` shell; the `||` means "run the commands between the {…} delimiters—the `echo` and the `exit`—only if the `jpackage` command fails to build an installer." Much of the complexity in the full version has to do with sorting out the OS-dependent parts and handling some options I added to make the script more flexible.

Once tamed by a script like mine, `jpackage` can easily and repeatably build installers for all three major desktop operating systems.

The `jlink` command predates `jpackage` and was used to create the runtime JDK image. Nowadays it is used directly from within `jpackage`; I no longer find it useful to call `jlink` except for use with `jpackage`.

This recipe is primarily concerned with producing application packages for desktop or traditional server-style applications. To package a REST or microservice application for deployment in a web server, just use `mvn deploy` as shown in Recipe 15.7. To package a Java REST or microservice app for Docker, you can use most of the frameworks listed in Table 15-1.

CHAPTER 3

Strings and Things

3.0 Introduction

Character strings are an inevitable part of just about any programming task. We use them for printing messages for the user; for referring to files on disk or other external media; and for people's names, addresses, and affiliations. The uses of strings are many, almost without number (actually, if you need numbers, we'll get to them in Chapter 5).

If you're coming from a programming language that is at or below C level, you'll need to remember that `String` is a defined type (class) in Java—that is, a string is an object and therefore has methods. It is not an array of characters (though it contains one) and should not be thought of as an array. Operations like `fileName.endsWith(".gif")` and `extension.equals(".gif")` (and the almost-equivalent `".gif".equals(extension)`) are commonplace.[1]

Java old-timers should note that Java 11 and 12 added several new `String` methods, including `indent(int n)`, `stripLeading()`/`stripTrailing()`, `Stream<String> lines()`, `isBlank()`, `repeat(int n)`, and `transform()`. Java 15 added `stripIndent()`. Most of these provide obvious functionality; `transform` allows applying an instance of a functional interface (see 9.0 Introduction) to a string and returning the result of that operation.

Although we haven't discussed the details of the java.io package yet (we will, in Chapter 10), you need to be able to read text files for some of these programs. Even if you're not familiar with `java.io`, you can probably see from the examples of reading

1 The two `.equals()` calls are equivalent with the exception that the first can throw a `NullPointerException` while the second cannot.

text files that a `BufferedReader` allows you to read chunks of data and that this class has a very convenient `readLine()` method.

Going the other way, `System.out.println()` is normally used to print strings or other values to the terminal or standard output. String concatenation is commonly used here, like this:

```
System.out.println("The answer is " + result);
```

One caveat with string concatenation is that if you are appending a bunch of values, and a number and a `char` (single character) are concatenated together, they are *added before concatenation* due to Java's precedence rules. So don't do as I did in this contrived example:

```
int result = ...;
System.out.println(result + '=' + " the answer."); // FAILURE
```

Given that `result` is an integer, then `result + =` (`result` added to the equals sign character, which is of the numeric type `char`) is a valid *numeric* expression, which will result in a single value of type `int`. If the variable `result` has the value 42, and given that the character = in a Unicode (or ASCII) code chart has the value 61, the two-line fragment would print:

```
103 the answer.
```

The wrong value and no equals sign! Safer approaches include using parentheses, double quotes around the equals sign, a `StringBuilder` (see Recipe 3.3), or `String.format()` (see Recipe 3.2). Of course in this simple example you could just move the = to be part of the string literal, but the example was chosen to illustrate the problem of arithmetic on `char` values being confused with string contatenation.

I won't show you how to sort an array of strings here; the more general notion of sorting a collection of objects will be taken up in Recipe 7.11.

String literals are *interned*, that is, they are stored internally. Interning is the process of storing only one copy of each unique string value in a pool of strings, as part of the class file containing the string. When a `String` is created at runtime, Java first checks if an equivalent string is present in the pool; if so, the existing string reference is returned instead of creating a new `String` object. This saves memory by avoiding duplicate string objects with the same value. The file *main/src/main/java/strings/Equality.java* demonstrates the results of interning, and a method of requesting it.

14 Java 14 enables text blocks, also known as multiline text strings. These are delimited with a set of three double quotes, the opening of which *must* have a newline after the quotes (which doesn't become part of the string; the following newlines do):

```
String longStr = """
This is a long
text String.""";
```

If we print `longStr`, we get:

```
This is a long
   text String.
```

Timeless, Immutable, and Unchangeable

Notice that a given `String` object, once constructed, is immutable. In other words, once I have said `String s = "Hello" + yourName;`, the contents of the particular object that reference variable `s` refers to can never be changed. You can assign `s` to refer to a different string, even one derived from the original, as in `s = s.trim()`. And you can retrieve characters from the original string using `charAt()`, but it isn't called `getCharAt()` because there is not, and never will be, a `setCharAt()` method. Even methods like `toUpperCase()` don't change the `String`; they return a new `String` object containing the translated characters. If you need to change characters within a `String`, you should instead create a `StringBuilder` (possibly initialized to the starting value of the `String`), manipulate the `StringBuilder` to your heart's content, and then convert that to `String` at the end, using the ubiquitous `toString()` method.[2]

How can I be so sure they won't add a `setCharAt()` method in the next release? Because the immutability of strings is one of the fundamentals of the Java Virtual Machine (JVM). Immutable objects are generally good for software reliability (some languages do not even allow mutable objects). Immutability avoids conflicts, particularly where multiple threads are involved, or where software from multiple organizations has to work together; for example, you can safely pass immutable objects to a third-party library and expect that the objects will not be modifed.

It may be possible to tinker with the `String`'s internal data structures using native code, as shown in Recipe 18.6, but then all bets are off.

Remember also that the `String` is a fundamental type in Java. Unlike most of the other classes in the core API, the behavior of strings is not changeable by subclassing; the class is marked `final` so it cannot be subclassed. So you can't declare your own `String` subclass. Think if you could—you could masquerade as a `String` but provide a `setCharAt()` method! Again, they thought of that. If you don't believe me, try it out:

```
public class WolfInStringsClothing
  extends java.lang.String {    //EXPECT COMPILE ERROR
  private static final long serialVersionUID = 1;

  public void setCharAt(int index, char newChar) {
    // The implementation of this method
    // would be left as an exercise for the reader.
```

2 `StringBuilder` was added in Java 5. It is functionally equivalent to the older `StringBuffer`. We will delve into the details in Recipe 3.3.

```
        // Hint: compile this code exactly as is before bothering!
    }
}
```

Got it? They thought of that!

Of course you do need to be able to modify strings. Some methods extract part of a `String`; these are covered in the first few recipes in this chapter. And `StringBuilder` is an important class that deals in characters and strings and has many methods for changing the contents, including, of course, a `toString()` method. Reformed C programmers should note that Java strings are not arrays of `chars` as in C. Therefore, you must use methods for such operations as processing a string one character at a time; see Recipe 3.4. Figure 3-1 shows an overview of `String`, `StringBuilder`, and C-language strings.

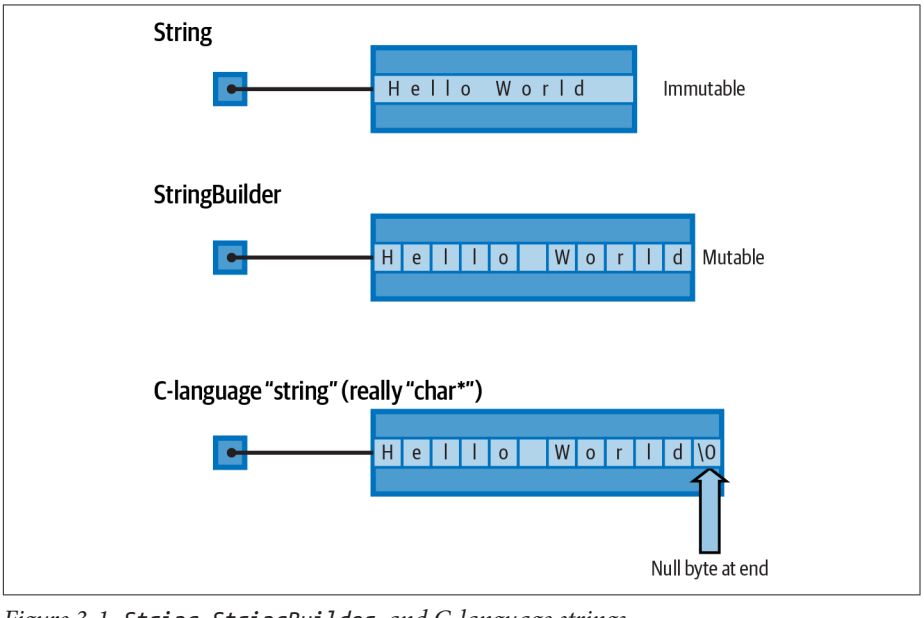

String
H e l l o W o r l d
Immutable
StringBuilder
H e l l o W o r l d
Mutable
C-language "string" (really "char*")
H e l l o W o r l d \0
Null byte at end

Figure 3-1. `String`, `StringBuilder`, and C-language strings

3.1 Taking Strings Apart with Substrings, Tokenizing, and Trimming Methods

Problem

You want to break a string apart, either by indexing positions or by using fixed token characters (e.g., break on spaces to get words).

Solution

For substrings, use the `String` object's `substring()` method. For tokenizing, construct a `StringTokenizer` around your string and call its methods `hasMoreTokens()` and `nextToken()`.

Or, use regular expressions (see Chapter 4).

Discussion

We'll look first at substrings, and then discuss tokenizing.

Substrings

The `substring()` method constructs a new `String` object made up of a run of characters contained somewhere in the original string, the one whose `substring()` you called. The `substring()` method is overloaded: both forms require a starting index (which is always *zero-based*). The one-argument form returns from `startIndex` to the end. The two-argument form takes an ending index (not a length, as in some languages) so that an index can be generated by the `String` methods `indexOf()` or `lastIndexOf()`:

```
public class SubStringDemo {
  public static void main(String[] av) {
    String a = "Java is great.";
    System.out.println(a);
    String b = a.substring(5);  // b is the String "is great."
    System.out.println(b);
    String c = a.substring(5,7);// c is the String "is"
    System.out.println(c);
    String d = a.substring(5,a.length());// d is "is great."
    System.out.println(d);
  }
}
```

When run, this prints the following:

```
C:> java SubStringDemo.java
Java is great.
is great.
is
is great.
C:>
```

Note that the end index is one beyond the last character! Java adopts this policy of having a half-open interval (or inclusive start, exclusive end) fairly consistently. There are good practical reasons for adopting this approach, and some other languages do so too.

Tokenizing

One of the easiest ways to split a string into words is to use the `String.split()` method. You can easily split a string into words like this:

```
for (String word : "Hello World".split(" ")) {
    System.out.println(word);
}
Hello
World
```

The parameter is actually a regular expression, discussed in Chapter 4. As a result, you can match multiple spaces, or spaces and tabs, by using the string `"\s+"`.

If you want to split the lines in a CSV file, you could try the string `","` but it's surely safer to use one of several third-party libraries for CSV files (because of issues like escaped commas in quoted string values).

Another method is to use a `StringTokenizer`. The `StringTokenizer` methods implement the `Iterator` interface and design pattern (see Recipe 7.7). However, its usage is officially "discouraged," but you will find it used in existing code:

main/src/main/java/strings/StrTokDemo.java

```
StringTokenizer st = new StringTokenizer("Hello World of Java");

while (st.hasMoreTokens( ))
    System.out.println("Token: " + st.nextToken( ));
```

The `split()` method mentioned previously is usually a better choice.

Sometimes you don't need to take the string completely apart, but simply remove leading or trailing blanks. For this, use the `String` class `strip()` or `trim()` methods.

There are five methods in the `String` class for this:

`String trim()`
: Returns the string with all leading and trailing space characters removed

`strip()`
: Returns the string with all leading and trailing whitespace removed

`stripLeading()`
: Returns the string with all leading whitespace removed

`stripTrailing()`
: Returns the string with all trailing whitespace removed

`stripIndent()`
: Returns the multiline string with indendation minimized

For the `trim()` method, space includes any character whose numeric value is less than or equal to 32, 0x20, or U+0020 (the actual space character). For the `strip()` methods, whitespace is as defined by `Character.isSpace()`, i.e., spaces, tabs, form feeds, etc.

See Also

Many occurrences of `StringTokenizer` can be replaced with regular expressions (see Chapter 4) with considerably more flexibility. For example, to extract all the numbers from a `String`, you can use this code:

```
Matcher tokenizer = Pattern.compile("\\d+").matcher(inputString);
while (tokenizer.find( )) {
        String courseString = tokenizer.group(0);
        int courseNumber = Integer.parseInt(courseString);
        ... do computation with courseNumber, or just print it ...
```

This allows user input to be more flexible than you could easily handle with a `StringTokenizer`. Assuming that the numbers represent course numbers at some educational institution, the inputs "471,472,570," or "Courses 471 and 472, 570," or just "471 472 570" all give the same results.

3.2 String Formatting with Formatter and printf()

Problem

You want to have fine-grained control over formatting values into strings, in such contexts as:

- The static `String.format()` method
- The Java 15 `string.formatted()` instance method
- The `printf()` method in the `PrintStream` and `PrintWriter` classes

Solution

Use the formatting capabilities of the `java.util.Formatter` class, which is used internally by at least the APIs listed previously.[3]

3 In addition to its use as a parameter in the `java.util.Formattable` interface, and not counting a dozen or so JDK-internal classes, `Formatter` is used by these classes: `java.net.HostPortrange`, `java.io.PrintWriter`, `java.io.PrintStream`, `java.lang.String`, and `java.util.logging.SimpleFormatter`.

Discussion

The `java.util.Formatter` class uses format codes derived from the C library's `printf()` function.[4] `PrintStream` and `PrintWriter` have convenience routines named `printf()` that simply delegate to the stream or writer's `format()` method, which in turn delegates to a `Formatter` instance. Unlike C, however, Java is a strongly typed language, so invalid arguments will throw an exception rather than generating gibberish. There are also convenience routines `static String.format()` and instance `string.formatted()` for use when you want to format a `String` without creating a `Formatter` explicitly and/or don't want it sent directly to an output stream or writer. You can even create a `Formatter` instance with a filename; this overload returns a `BufferedWriter` whose `format()` methods will write into the file. Close that when done, or use in a `try-with-resources` to automatically close it.

There are many classes to perform formatting in Java. Do not confuse `java.util.Formatter` with `java.text.Format`. `Formatter` is for `printf`-like formatting of one or more values with control over formatting. `Format` is a base class with subclasses for formatting dates, strings, numbers, lists, and more, all in a locale-sensitive way. `Format` has a subclass `MessageFormat`, which uses curly-brace markers to interpolate parameters into a string. `Format` is also the base class of my `StringAlign` class (see Recipe 3.5) and my `RomanNumberFormat` class (discussed in a subsection of Recipe 5.5).

The underlying `java.util.Formatter` class works on a "format string" containing format codes. For each item that you want to format, you put a format code into the format string, and pass a parameter to be formattted according to that code. The format code consists of a percent sign, optionally an argument number followed by a dollar sign, optionally a field width or precision, and a format type (e.g., `d` for decimal integer, that is, an integer with no decimal point, and `f` for floating point).

One of the simplest forms is when you just need to format a string:

```
var str = String.format("Hello %s, today is %tF%n", name, now);
```

15 Some developers (and some IDEs) prefer to provide a string containing the format codes, using the instance method `formatted`. The following provides the same formatting result as the previous code:

```
var str2 = "Hello %s, today is %tF%n".formatted(name, now);
```

4 Designed in the early 1970s by Mike Lesk at AT&T Bell Labs, it first saw widespread use in the library for the C language Dennis Ritchie had just invented, and it has been included or available in almost every programming language since. It has also, inevitably, grown significantly over time.

This form is useful in internationalization (see Recipe 3.11) when you want to provide different strings for use in different locales. However, it is *strongly* recommended to avoid using user-provided format strings—those you might have read in from a user, especially over the internet—since the language that `format()` accepts is such a broad attack surface from a security point of view.

In any JDK in living memory, a simple usage that combines formatting with output might look like the following:

```
// To output the formatted result:
System.out.printf("%04d - the year of %f%n", 1956, Math.PI);
// or, more formally:
System.out.printf("1$%04d - the year of %2$f%n", 1956, Math.PI);
```

As shown in Figure 3-2, the `"%04d"` or `"%1$04d"` controls formatting of the year, and the `"%f"` or `"%2$f"` controls formatting of the value of `PI`.[5]

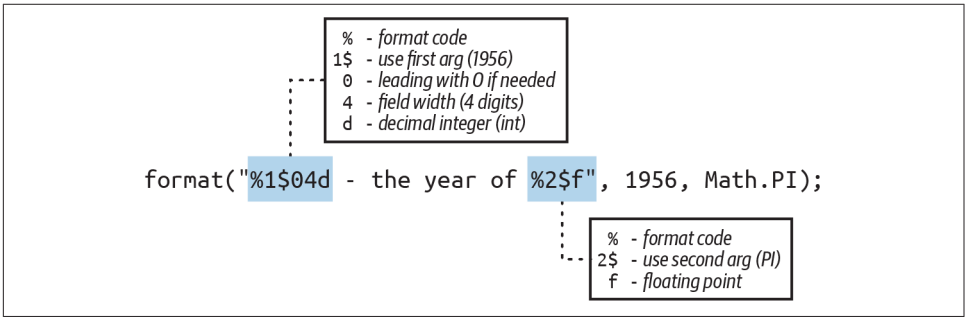

% - format code
1$ - use first arg (1956)
0 - leading with 0 if needed
4 - field width (4 digits)
d - decimal integer (int)
format("%1$04d - the year of %2$f", 1956, Math.PI);
% - format code
2$ - use second arg (PI)
f - floating point

Figure 3-2. `Formatter` codes examined

Many format codes are available; Table 3-1 lists some of the more common ones. There is considerably more detail and more choices in the Javadoc for `java.util.Formatter`.

Table 3-1. `Formatter` format codes

Code	Meaning
b	Boolean true or false.
c	Character (argument must be `char` or integral type containing valid character value).
d	decimal int—integer to be printed as a decimal (radix 10) with no decimal point (argument must be integral type).
e	Floating-point value printed in scientific ("exponential") notation.

5 The central character in Yann Martel's novel *Life of Pi* would have been born in 1956, according to information in Wikipedia.

Code	Meaning
f	Floating-point value with decimal fraction (must be numeric); field width may be followed by decimal point and fractional digit field width; e.g., 7.2f.
g	Floating-point value, as per f or e, depending on magnitude.
h	Print object's hashCode() in hex, as per Integer.toHexString(arg.hashCode()).
s	Generic format; if value is null, prints null; else if arg implements Formattable, format as per arg.formatTo(); else format as per arg.toString().
t	Date codes; follow with secondary code. Common date codes are shown in Table 3-2. Argument must be long, Long, LocalDate and friends, or the legacy Calendar or java.util.Date.
n	Newline; insert the platform-dependent line ending character.
%	Insert a literal % character.

The letters b, c, e, h, s may be entered in uppercase, in which case the result will be capitalized as appropriate.

Note also that you may, but are not required to, put a *parameter order* number between the % and the format code. For example, in "%2$04d", the "2$" means to format the *second* parameter, regardless of the order of the parameters. This is primarily useful with dates (see the following example, where you need to format several different portions of the same Date or Calendar, or any time you want to format the same object more than once) and in internationalization, where different languages may require words to be in a different order within a sentence.

Some examples of using a Formatter are shown in Example 3-1.

Example 3-1. main/src/main/java/io/FormatterDemo.java

```
public class FormatterDemo {
  public static void main(String[] args) throws IOException {

    // The arguments to all these format methods consist of
    // a format code String and 1 or more arguments.
    // Each format code consists of the following:
    // % - code lead-in
    // N$ - OPTIONAL parameter number (1-based) after the format code
    // N - field width
    // L - format letter (d: decimal(int); f: float; s: general; many more)

    // Most general (cumbersome) way of proceeding.
    Formatter fmtr = new Formatter();
    // The format() method returns a Formatter, so use Object
    Object result = fmtr.format("%1$04d - the year of %2$f", 1956, Math.PI);
    System.out.println(result);

    // Shorter way using static String.format(), default parameter numbering.
    String stringResult = "%04d - the year of %f".formatted(1956, Math.PI);
    System.out.println(stringResult);
```

```
    // A shorter way using PrintStream/PrintWriter.format, more in line with
    // other languages. Using this way, you should provide the newline delimiter
    // using %n (rather than \n as that is platform-dependent!).
    System.out.printf("%04d - the year of %f%n", 1956, Math.PI);

    // Format doubles with more control
    System.out.printf("PI is approximately %4.2f%n", Math.PI);

    final String TMPFILE = "/tmp/format-demo.txt";
    Formatter fmt2 = new Formatter(TMPFILE);
    fmt2.format(
      "This is a %s opportunity that you can get in for just $%5.2f!%n",
      "new", 27.99);
    fmt2.close();

    String result2 = Files.readString(Path.of(TMPFILE));
    System.out.print("String from file is: " + result2);
  }
}
```

Running `FormatterDemo` produces this:

```
C:> java FormatterDemo.java
1956 - the year of 3.141593
1956 - the year of 3.141593
1956 - the year of 3.141593
PI is approximately 3.14
String from file is: This is a new opportunity that you can get in for just
$27.99!
C:>
```

For formatting date and time objects, a large variety of format codes is available—about 40 in all. Date and time objects are discussed in Chapter 6. Table 3-2 shows the more common date/time format codes. Each must be preceded by a `t`, so to format the first argument as a year, you would use `%1$tY`.

Table 3-2. Formatting codes for dates and times

Format code	Meaning
`A`	Locale-specific day of week (`a` for abbreviated)
`B`	Locale-specific month name (`b` for abbreviated)
`d`	Day of month (two digits, leading zeros)
`e`	Day of month (one or two digits)
`H` or `I`	Hour in 24-hour (`H`) or 12-hour (`I`) format (two digits, leading zero if needed)
`M`	Minute (two digits)
`m`	Month as two-digit (leading zeros) number
`P`/`p`	Locale-specific AM or PM in uppercase (if `P`) or lowercase (if `p`)

Format code	Meaning
S	Second (two digits)
y	Two-digit year (for those who don't remember Y2K)
Y	Year (at least four digits)

There are also some composite date codes, including those in Table 3-3.

Table 3-3. Composite formatting codes for dates and times

Format code	Meaning
F	ISO 8601 standard date formatted as `"%tY-%tm-%td"`
D	US-only date format as `"%tm/%td/%ty"`
R or T	24-hour time combination: `%tH:%tM` (if R) or `%tH:%tM:%tS` (if T)
c	Fancy date string, requires `java.util.Date` or `ZonedDateTime`, not `LocalDateTime`

In my opinion, embedding the codes from Table 3-2 directly into applications that you distribute or make available as web applications is often a bad idea. Any direct use of them assumes that you know *the correct order to print these fields in all locales around the world*. Trust me, you don't. Instead of these, I recommend the use of `DateTimeFormatter`, covered in Recipe 6.2, to control the order of arguments. However, for quick-and-dirty work, as well as for writing log- or datafiles that must be in a given format because some other program reads them, these are OK. Some date examples are shown in Example 3-2.

Example 3-2. main/src/main/java/io/FormatterDates.java

```
public class FormatterDates {
  public static void main(String[] args) {

    // Format number as dates e.g., 2026-06-28
    System.out.printf("%4d-%02d-%2d%n", 2026, 6, 28);

    // Format fields directly from a Date object: multiple fields from "1$"
    // (hard-coded formatting for Date not advisable; see I/O chapter)
    LocalDate today = LocalDate.now();
    // Print in a form like e.g., "July 4, 2026"
    System.out.printf("Today is %1$tB %1$td, %1$tY%n", today);
  }
}
```

When using a date-related object to format multiple fields, as in this example, the `"1$"` (or appropriate parameter number, `"2$"`, `"3$"`, etc.) must be specified for each format code.

Running this `FormatterDates` class produces the following output:

```
$ java FormatterDates.java
2027-01-02
Today is January 02, 2027
$
```

String templates 21P

Java 21 and 22 introduced a preview feature `StringTemplate` class, which allowed one to interpolate variables into a string using `\{varName}`:

```
var str3 = STR."Hello \{name}, today is \{now}";
```

Sadly, Oracle withdrew the `StringTemplate` preview from Java 23, pending the creation of a better way of implementing string interpolation. The latest word on this is in JEP 465, which is now marked "closed/withdrawn." This JEP, like most, does have a good discussion of why the feature is needed and how they arrived at the design. Apparently the design did not pan out when used in "a substantial, real-world internal project" (Oracle's Gavin Bierman and Brian Goetz, *https://mail.openjdk.org/pipermail/amber-spec-experts/2024-April/004106.html*) so the feature was withdrawn until a better solution can be found.

3.3 Building Strings with StringBuilder

Problem

You need to put some `String` pieces (back) together.

Solution

Use string concatenation: the `+` operator. The compiler implicitly constructs a `StringBuilder` for you and uses its `append()` methods (unless all the string parts are known at compile time).

Better yet, construct and use a `StringBuilder` yourself.

Discussion

An object of the `StringBuilder` class basically represents a collection of characters. It is similar to a `String` object.[6] However, as mentioned, `String`s are immutable; `StringBuilder`s are mutable and designed for, well, building `String`s. You typically

6 `String` and `StringBuilder` have several methods that are forced to be identical by their implementation of the `CharSequence` interface.

construct a `StringBuilder`, invoke the methods needed to get the character sequence just the way you want it, and then call `toString()` to generate a `String` representing the same character sequence for use in most of the Java API, which deals in `String`s.

A similar class, `StringBuffer`, is historical—it's been around since the beginning of time. Some of its methods are synchronized (see Recipe 11.4), which involves unneeded overhead in a single-threaded context. In Java 5, this class was split into `StringBuffer` (which is synchronized) and `StringBuilder` (which is not synchronized); thus, it is faster and preferable for single-threaded use. Another new class, `AbstractStringBuilder`, is the parent of both. In the following discussion, I'll use "the `StringBuilder` classes" to refer to all three because they mostly have the same methods.

The book's example code provides a `StringBuilderDemo` and a `StringBufferDemo`. Except for the fact that `StringBuilder` is not thread-safe, these API classes are identical and can be used interchangeably, so my two demo programs are almost identical except that each one uses the appropriate builder class.

The `StringBuilder` classes have a variety of methods for inserting, replacing, and otherwise modifying a given `StringBuilder`. Conveniently, the `append()` methods return a reference to the `StringBuilder` itself, so stacked statements like `.append(…).append(…)` are fairly common. This style of coding is referred to as a *fluent API* because it reads smoothly, like prose from a native speaker of a human language. You might even see this style of coding in a `toString()` method, for example. Example 3-3 shows three ways of concatenating strings.

Example 3-3. main/src/main/java/strings/StringBuilderDemo.java

```
public class StringBuilderDemo {

  public static void main(String[] argv) {

    String s1 = "Hello" + ", " + "World";
    System.out.println(s1);

    // Build a StringBuilder, and append some things to it.
    StringBuilder sb2 = new StringBuilder();
    sb2.append("Hello");
    sb2.append(',');
    sb2.append(' ');
    sb2.append("World");

    // Get the StringBuilder's value as a String, and print it.
    String s2 = sb2.toString();
    System.out.println(s2);

    // Now do the above all over again, but in a more
    // concise (and typical "real-world" Java) fashion.
```

```
    System.out.println(
      new StringBuilder()
      .append("Hello")
      .append(',')
      .append(' ')
      .append("World"));
  }
}
```

In fact, all the methods that modify the characters in a `StringBuilder` (i.e., `append()`, `delete()`, `deleteCharAt()`, `insert()`, `replace()`, and `reverse()`) return a reference to the builder object to facilitate this fluent API style of coding.

As another example of using a `StringBuilder`, consider the need to convert a list of items into a comma-separated list while avoiding getting an extra comma after the last element of the list. This can be done using a `StringBuilder`, although in Java 8+ there is a static `String` method to do the same. Code for these are shown in Example 3-4.

Example 3-4. main/src/main/java/strings/StringBuilderCommaList.java

```
    System.out.println(
      "Split using String.split; joined using 1.8 String join");
    System.out.println(String.join(", ", SAMPLE_STRING.split(" ")));

    System.out.println(
      "Split using String.split; joined using StringBuilder");
    StringBuilder sb1 = new StringBuilder();
    for (String word : SAMPLE_STRING.split(" ")) {
      if (sb1.length() > 0) {
        sb1.append(", ");
      }
      sb1.append(word);
    }
    System.out.println(sb1);
```

The first method is clearly the most compact; the static `String.join()` makes short work of this task. The next method uses the `StringBuilder.length()` method, so it will only work correctly when you are starting with an empty `StringBuilder`. A third method, in the online source but not printed here, involves a `StringTokenizer` and the information method `hasMoreElements()` in the `Enumeration`. An alternative method, particularly when you aren't starting with an empty builder, would be to use a `boolean` flag variable to track whether you're at the beginning of the list.

3.4 Processing a String One Character at a Time

Problem

You want to process the contents of a string, one character at a time.

Solution

Use a `for` loop and the `String`'s `charAt()` or `codePointAt()` method. Or use a `for each` loop and the `String`'s `toCharArray` method.

Discussion

A string's `charAt()` method retrieves a given character by index number (starting at zero) from within the `String` object. Since Unicode has had to expand beyond 16 bits, not all Unicode characters can fit into a Java `char` variable. There is thus an analogous `codePointAt()` method, whose return type is `int`. To process all the characters in a `String`, one after another, use a `for` loop ranging from zero to `String.length()-1`. Here we process all the characters in a `String`:

main/src/main/java/strings/strings/StrCharAt.java

```
public class StrCharAt {
    public static void main(String[] av) {
        String a = "A quick bronze fox";
    for (int i=0; i < a.length(); i++) { // no forEach, need the index
      String message =
        "charAt is '%c', codePointAt is %3d, casted it's '%c'".formatted(
        a.charAt(i),
        a.codePointAt(i),
        (char)a.codePointAt(i));
      System.out.println(message);
    }
  }
}
```

Given that the `for each` loop has been in the language for ages, you might be excused for expecting to be able to write some code like `for (char ch : myString) {…}`. Unfortunately, this does not work, as the `String` class is not iterable (maybe it should be, but it's complicated). You can use `myString.toCharArray()`, which does use some memory for the new array, as in the following:

```
public class ForEachChar {
  public static void main(String[] args) {
    String mesg = "Hello world";

    // Does not compile, Strings are not iterable
    // for (char ch : mesg) {
```

```
        //    System.out.println(ch);
        // }

        System.out.println("Using toCharArray:");
        for (char ch : mesg.toCharArray()) {
          System.out.println(ch);
        }

        System.out.println("Using Streams:");
        mesg.chars().forEach(c -> System.out.println((char)c));
      }
    }
```

A *checksum* is a numeric quantity representing and confirming the contents of a file. If you transmit the checksum of a file separately from the contents, a recipient can checksum the file—assuming the algorithm is known—and verify that the file was received intact. Example 3-5 shows the simplest possible checksum, computed just by adding the numeric values of each character. Note that on files, it does not include the values of the newline characters; in order to fix this, retrieve `System.getProperty("line.separator");` and add its character value(s) into the sum at the end of each line. Or give up on line mode and read the file a character at a time.

Example 3-5. main/src/main/java/strings/CheckSum.java

```
/** Checksum one text file, given an open BufferedReader.
 * Checksum does not include line endings, so it will give the
 * same value for given text on any platform. Do not use
 * on binary files!
 */
public static int process(BufferedReader is) {
    int sum = 0;
    try {
        String inputLine;

        while ((inputLine = is.readLine()) != null) {
            for (char c : inputLine.toCharArray()) {
                sum += c;
            }
        }
    } catch (IOException e) {
        throw new RuntimeException("IOException: " + e);
    }
    return sum;
}
```

3.5 Aligning, Indenting, and Unindenting Strings

Problem

You want to align strings to the left, right, or center.

Solution

Do the math yourself, and use `substring` (see Recipe 3.1) and a `StringBuilder` (see Recipe 3.3). Or, use my `StringAlign` class, which is based on the `java.text.Format` class. For left or right alignment, use `String.format()`.

Discussion

Centering and aligning text comes up fairly often. Suppose you want to print a simple report with centered page numbers. There doesn't seem to be anything in the standard API that will do the job fully for you. But I have written a class called `StringAlign` that will. Here's how you might use it:

```
public class StringAlignSimple {

  public static void main(String[] args) {
    // Construct a "formatter" to center strings.
    StringAlign formatter = new StringAlign(70, StringAlign.Justify.CENTER);
    // Try it out, for page "i"
    System.out.println(formatter.format("- i -"));
    // Try it out, for page 4. Since this formatter is
    // optimized for Strings, not specifically for page numbers,
    // we have to convert the number to a String
    System.out.println(formatter.format(Integer.toString(4)));
  }
}
```

If you run this class, it prints the two demonstration line numbers centered, as shown:

```
C:\> java StringAlignSimple.java
                                    - i -
                                      4
C:\>
```

The method `String.repeat()` returns, as its name implies, a string that contains *n* copies of the input:

```
jshell> var response = "Aye ".repeat(2)
response ==> "Aye Aye "
jshell>
```

Example 3-6 is the code for my `StringAlign` class. Note that this class extends the class `Format` in the package java.text. There is a series of `Format` classes that all have at

least one method called `format()`. It is thus in a family with numerous other formatters, such as `DateFormat` and `NumberFormat`, which we'll take a look at in upcoming chapters.

Example 3-6. main/src/main/java/strings/StringAlign.java

```
public class StringAlign extends Format {

  private static final long serialVersionUID = 1L;

  public enum Justify {
    /* Constant for left justification. */
    LEFT,
    /* Constant for centering. */
    CENTER,
    /** Constant for right-justified Strings. */
    RIGHT,
  }

  /** Current justification */
  private Justify just;
  /** Current max length */
  private int maxChars;

    /** Construct a StringAlign formatter; length and alignment are
     * passed to the Constructor instead of each format() call as the
     * expected common use is in repetitive formatting e.g., page numbers.
     * @param maxChars - the maximum length of the output
     * @param just - one of the enum values LEFT, CENTER, or RIGHT
     */
  public StringAlign(int maxChars, Justify just) {
    switch(just) {
    case LEFT:
    case CENTER:
    case RIGHT:
      this.just = just;
      break;
    default:
      throw new IllegalArgumentException("invalid justification arg.");
    }
    if (maxChars < 0) {
      throw new IllegalArgumentException("maxChars must be positive.");
    }
    this.maxChars = maxChars;
  }

  /** Format a String.
   * If the input is too long, it will be returned, truncated.
   * @param input - the string to be aligned.
   * @parm where - the StringBuilder to append it to.
   * @param ignore - a FieldPosition (may be null, not used but
```

```
     * specified by the general contract of Format).
     * @return A StringBufffer with the formatted string; sadly not
     * the newer StringBuilder because the old interface is cast in stone.
     */
    @Override
    public StringBuffer format(
      Object input, StringBuffer where, FieldPosition ignore)  {

      String s = input.toString();
      String wanted = s.substring(0, Math.min(s.length(), maxChars));

      // Get the spaces in the right place.
      switch (just) {
        case RIGHT:
          pad(where, maxChars - wanted.length());
          where.append(wanted);
          break;
        case CENTER:
          int toAdd = maxChars - wanted.length();
          pad(where, toAdd/2);
          where.append(wanted);
          pad(where, toAdd - toAdd/2);
          break;
        case LEFT:
          where.append(wanted);
          pad(where, maxChars - wanted.length());
          break;
        }
      return where;
    }

    protected final void pad(StringBuffer to, int howMany) {
      for (int i=0; i<howMany; i++)
        to.append(' ');
    }

    /** Convenience Routine */
    String format(String s) {
      return format(s, new StringBuffer(), null).toString();
    }

    /** ParseObject is required, but not useful here. */
    public Object parseObject(String source, ParsePosition pos)  {
      return source;
    }
}
```

A number of methods in `String` work on multiline strings, those with embedded newline characters. These can be produced in many ways, including multiline text strings (see Java 14 Text Blocks) or read from a file; for example, using `Files.readString()`. Java 12 introduced a new method, `public String indent(int n)`, that

prepends *n* spaces to *each line* of the given multiline string. If *n* is negative, the number of spaces is removed from the front of each line. This works well in conjunction with the Java 11 `Stream<String> lines()` String instnce method. For example, for the case where a series of lines, already stored in a single string, needs the same indent (`Streams`, and the "::" notation, are explained in Recipe 9.8):

```
jshell> "abc\ndef".indent(30).lines().forEach(System.out::println);
                              abc
                              def

jshell> "abc\ndef".indent(30).indent(-10).lines().forEach(System.out::println);
                    abc
                    def

jshell>
```

`stripIndent()` finds and removes the smallest leading whitespace indentation in a multiline string, preserving relative indentation within the string:

```
jshell> var input = """
   ...>     The following:
   ...>        Volume 1
   ...>        Volume 4
   ...>           4A
   ...>           4B"""
input ==> "The following:\n   Volume 1\n   Volume 4\n      4A\n      4B"

jshell> System.out.println(input.stripIndent());
The following:
   Volume 1
   Volume 4
      4A
      4B

jshell>
```

In this example, four spaces have been removed from the front of each line.

See Also

The alignment of numeric columns is considered in Chapter 5.

3.6 Converting Between Unicode Characters and Strings

Problem

You want to convert between Unicode characters and `Strings`.

Solution

Use Java `char` or `String` data types to deal with characters; these intrinsically support Unicode. Print characters as integers to display their *raw* value if needed.

Discussion

Unicode is an international standard that aims to represent all known characters used by people in all their various languages. Though the original ASCII character set is a subset, Unicode is huge. At the time Java was created, Unicode was a 16-bit character set, so it seemed natural to make Java `char` values be 16 bits in width, and for years a `char` could hold any Unicode character. However, over time, Unicode has grown, to the point that it now includes over a million code points, or characters, more than the 65,525 that could be represented in 16 bits.[7] Not all possible 16-bit values were defined as characters in UCS-2, the 16-bit version of Unicode originally used in Java. A few were reserved as escape characters, which allows for multicharacter-length mappings to less common characters. Fortunately, there is a go-between standard, called UTF-16 (16-bit Unicode Transformation Format). As the `String` class documentation puts it:

> A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section Unicode Character Representations in the `Character` class for more information). Index values refer to char code units, so a supplementary character uses two positions in a `String`.
>
> The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., `char values`).

The `charAt()` method of `String` returns the `char` value for the character at the specified offset. If you might have to deal with character sets with characters beyond the 16-bit range, use `codePointAt()` instead. The `StringBuilder append()` method has a form that accepts a `char`. Because `char` is an integer type, you can even do arithmetic on `char`s, though this is not needed as frequently as in, say, C. Nor is it often recommended, because the `Character` class provides the methods for which these operations were normally used in languages such as C. Here is a program that uses arithmetic on `char`s to control a loop and that also appends the characters into a `StringBuilder` (see Recipe 3.3):

```
// UnicodeChars.java
StringBuilder b = new StringBuilder();
for (char c = 'a'; c<'d'; c++) {
  b.append(c);
```

7 Indeed, there are so many characters in Unicode that a fad has emerged of displaying your name upside down using characters that approximate upside-down versions of the Latin alphabet. Do a web search for "upside-down Unicode."

```
}
b.append('\u00a5');  // Japanese Yen symbol
b.append('\u01FC');  // Roman AE with acute accent
b.append('\u0391');  // GREEK Capital Alpha
b.append('\u03A9');  // GREEK Capital Omega

for (int i=0; i<b.length(); i++) {
  System.out.printf(
    "Character #%d (%04x) is %c%n",
    i, (int)b.charAt(i), b.charAt(i));
}
System.out.println("Accumulated characters are " + b);
```

When you run it, the expected results are printed for the ASCII characters. On Unix and characters in Mac systems, the default fonts don't include all the additional characters, so they are either omitted or mapped to irregular characters:

```
$ java UnicodeChars.java
Character #0 (0061) is a
Character #1 (0062) is b
Character #2 (0063) is c
Character #3 (00a5) is ¥
Character #4 (01fc) is Ǽ
Character #5 (0391) is A
Character #6 (03a9) is Ω
Accumulated characters are abc¥ǼAΩ
$
```

The Windows system that I used to try this doesn't have most of those characters either, but at least it prints question marks for the ones it knows are lacking (Windows system fonts are more homogenous than those of the various Unix systems, so it may be easier to know what won't work):

```
$ java UnicodeChars.java
Character #0 is a
Character #1 is b
Character #2 is c
Character #3 is ¥
Character #4 is ?
Character #5 is ?
Character #6 is ?
Accumulated characters are abc¥...
```

The "." characters are unprintable characters.

Characters above `0xffff` need special handling; one way is to pass the numeric equivalents into the String constructor as an array of `int`s.

```
// Let's show Santa (0x1f385) + Star (0x2600):
int[] codePoints = {0x1f385, 0x2600};
String faceAndStar = new String(codePoints, 0, codePoints.length);
System.out.println(faceAndStar);    // "🎅☀"
```

See Also

The `Unicode` program in this book's online source displays any 256-character section of the Unicode character set. You can download documentation listing every character in the Unicode character set from the Unicode Consortium.

3.7 Reversing a String by Word or by Character

Problem

You wish to reverse a string, either character by character or word by word.

Solution

You can reverse a string by character easily, using a `StringBuilder`. There are several ways to reverse a string a word at a time. One natural way is to use a `StringTokenizer` and a stack. `Stack` is a class (defined in `java.util`; see Recipe 7.6) that implements an easy-to-use last-in, first-out (LIFO) stack of objects.

Discussion

To reverse the characters in a string, use the `StringBuilder reverse()` method:

main/src/main/java/strings/StringRevChar.java

```
String sh = "FCGDAEB";
System.out.println(sh + " -> " + new StringBuilder(sh).reverse());
```

The letters in this example list the order of the sharps in the key signatures of Western music; in reverse, it lists the order of flats. Alternatively, of course, you could reverse the characters yourself, using one-character-at-a-time mode (see Recipe 3.4).

A popular mnemonic, or memory aid, to help music students remember the order of sharps and flats consists of one word for each sharp instead of just one letter. Let's reverse this *one word at a time*. Example 3-7 adds each one to a `Stack` (see Recipe 7.6), then processes the whole lot in LIFO order, which reverses the order.

Example 3-7. main/src/main/java/strings/StringReverse.java

```
String sample = "Father Charles Goes Down And Ends Battle";

// An older way
// Put it in the stack frontward
Stack<String> myStack = new Stack<>();
var forward = sample.split("\\s");
for (String str : forward) {
  myStack.push(str);
```

```
}

// Print the stack backward
while (!myStack.empty()) {
  System.out.print(myStack.pop());
  System.out.print(' ');  // inter-word spacing
}
System.out.println();
```

This prints:

```
Battle Ends And Down Goes Charles Father
```

21 Java 21 added a default `reversed()` method to the `List` interface, and since Java 9 we could make a `List` from an array using an overload of `List.of()`. We can write this idiomatically in a single statement:

```
// The easier way (Java 21+, no temporary variables)
System.out.println(String.join(" ",
  List.of(sample.split("\\s")).reversed()));
```

This prints the list the same as Example 3-7. Note that `reversed()` creates a view onto the original list, so one needs to be careful about modifying the original while traversing the view.

3.8 Expanding and Compressing Tabs

Problem

You need to convert space characters to tab characters in a file, or vice versa. You might want to replace spaces with tabs to save space on disk or go the other way to deal with a device or program that can't handle tabs.

Solution

Use my `Tabs` class or its subclass `EnTab`.

Discussion

Because programs that deal with tabbed text or data expect tab stops to be at fixed positions, you cannot use a typical text editor to replace tabs with spaces or vice versa. Example 3-8 is a listing of `EnTab`, complete with a sample main program. The program works a line at a time. For each character on the line, if the character is a space, we see if we can coalesce it with previous spaces to output a single tab character. This program depends on the `Tabs` class, which we'll come to shortly. The `Tabs` class is used to decide which column positions represent tab stops and which do not.

Example 3-8. main/src/main/java/strings/Entab.java

```
public class EnTab {

  private static Logger logger = Logger.getLogger(EnTab.class.getSimpleName());

  /** The Tabs (tab logic handler) */
  protected Tabs tabs;

  /**
   * Delegate tab spacing information to tabs.
   */
  public int getTabSpacing() {
    return tabs.getTabSpacing();
  }

  /**
   * Main program: just create an EnTab object, and pass the standard input
   * or the named file(s) through it.
   */
  public static void main(String[] argv) throws IOException {
    EnTab et = new EnTab(8);
    if (argv.length == 0) // do standard input
      et.entab(
        new BufferedReader(new InputStreamReader(System.in)),
        System.out);
    else
      for (String fileName : argv) { // do each file
        et.entab(
          new BufferedReader(new FileReader(fileName)),
          System.out);
      }
  }

  /**
   * Constructor: just save the tab values.
   * @param n The number of spaces each tab is to replace.
   */
  public EnTab(int n) {
    tabs = new Tabs(n);
  }

  public EnTab() {
    tabs = new Tabs();
  }

  /**
   * entab: process one file, replacing blanks with tabs.
   * @param is A BufferedReader opened to the file to be read.
   * @param out a PrintWriter to send the output to.
   */
  public void entab(BufferedReader is, PrintWriter out) throws IOException {
```

```
    // main loop: process entire file one line at a time.
    is.lines().forEach(line -> {
      out.println(entabLine(line));
    });
  }

  /**
   * entab: process one file, replacing blanks with tabs.
   *
   * @param is A BufferedReader opened to the file to be read.
   * @param out A PrintStream to write the output to.
   */
  public void entab(BufferedReader is, PrintStream out) throws IOException {
    entab(is, new PrintWriter(out));
  }

  /**
   * entabLine: process one line, replacing blanks with tabs.
   * @param line the string to be processed
   */
  public String entabLine(String line) {
    int N = line.length(), outCol = 0;
    StringBuilder sb = new StringBuilder();
    char ch;
    int consumedSpaces = 0;

    for (int inCol = 0; inCol < N; inCol++) { // Cannot use foreach here
      ch = line.charAt(inCol);
      // If we get a space, consume it, don't output it.
      // If this takes us to a tab stop, output a tab character.
      if (ch == ' ') {
        logger.info("Got space at " + inCol);
        if (tabs.isTabStop(inCol)) {
          logger.info("Got a Tab Stop " + inCol);
          sb.append('\t');
          outCol += consumedSpaces;
          consumedSpaces = 0;
        } else {
          consumedSpaces++;
        }
        continue;
      }

      // We're at a non-space; if we're just past a tab stop, we need
      // to put the "leftover" spaces back out, since we consumed
      // them above.
      while (inCol-1 > outCol) {
        logger.info("Padding space at " + inCol);
        sb.append(' ');
        outCol++;
      }
```

```
      // Now we have a plain character to output.
      sb.append(ch);
      outCol++;

    }
    // If line ended with trailing (or only!) spaces, preserve them.
    for (int i = 0; i < consumedSpaces; i++) {
      logger.info("Padding space at end # " + i);
      sb.append(' ');
    }
    return sb.toString();
  }
}
```

This code was patterned after a program in Brian Kernighan and P.J. Plauger's classic work *Software Tools* (Addison-Wesley). While their version was in a language called RatFor (Rational Fortran), my version has since been through several translations. Their version actually worked one character at a time, and for a long time I tried to preserve this overall structure. Eventually, I rewrote it to be a line-at-a-time program.

The program that goes in the opposite direction—taking tabs out instead of putting them in—is the `DeTab` class shown in Example 3-9; only the core methods are shown.

Example 3-9. main/src/main/java/strings/DeTab.java

```
public class DeTab {
  Tabs ts;

  public static void main(String[] argv) throws IOException {
    DeTab dt = new DeTab(8);
    dt.detab(new BufferedReader(new InputStreamReader(System.in)),
        new PrintWriter(System.out));
  }

  public DeTab(int n) {
    ts = new Tabs(n);
  }
  public DeTab() {
    ts = new Tabs();
  }

  /** detab one file (replace tabs with spaces)
   * @param is - the file to be processed
   * @param out - the updated file
   */
  public void detab(BufferedReader is, PrintWriter out) throws IOException {
    is.lines().forEach(line -> {
      out.println(detabLine(line));
    });
  }
```

```
  /** detab one line (replace tabs with spaces)
   * @param line - the line to be processed
   * @return the updated line
   */
  public String detabLine(String line) {
    char c;
    int col;
    StringBuilder sb = new StringBuilder();
    col = 0;
    for (int i = 0; i < line.length(); i++) {
      // Either ordinary character or tab.
      if ((c = line.charAt(i)) != '\t') {
        sb.append(c); // Ordinary
        ++col;
        continue;
      }
      do { // Tab, expand it, must put >=1 space
        sb.append(' ');
      } while (!ts.isTabStop(++col));
    }
    return sb.toString();
  }
}
```

The `Tabs` class maintains the tab settings and provides the `istabstop()` method. Example 3-10 is the source for the `Tabs` class.

Example 3-10. main/src/main/java/strings/Tabs.java

```
public class Tabs {
  /** tabs every so often */
  public final static int DEFTABSPACE =   8;
  /** the current tab stop setting. */
  protected int tabSpace = DEFTABSPACE;
  /** The longest line that we initially set tabs for. */
  public final static int MAXLINE  = 255;

  /** Construct a Tabs object with a given tab stop setting */
  public Tabs(int n) {
    if (n <= 0) {
      n = 1;
    }
    tabSpace = n;
  }

  /** Construct a Tabs object with a default tab stop setting */
  public Tabs() {
    this(DEFTABSPACE);
  }
```

```
  /**
   * @return Returns the tabSpace.
   */
  public int getTabSpacing() {
    return tabSpace;
  }

  /** isTabStop - returns true if given column is a tab stop.
   * @param col - the current column number
   */
  public boolean isTabStop(int col) {
    if (col <= 0)
      return false;
    return (col+1) % tabSpace == 0;
  }
}
```

3.9 Controlling Case

Problem

You need to either convert strings to uppercase or lowercase or compare strings without regard for case.

Solution

The `String` class has a number of methods for dealing with documents in a particular case.

Discussion

String methods `toUpperCase()` and `toLowerCase()` each return a new string that is a copy of the current string but converted, as the name implies. Each can be called either with no arguments or with a `Locale` argument specifying the conversion rules; this is necessary because of internationalization. Java's API provides significant internationalization and localization features, as covered in "Ian's Basic Steps: Internationalization and Localization" on page 141. Whereas the `equals()` method tells you if another string is exactly the same, `equalsIgnoreCase()` tells you if all characters are the same regardless of case. Here, you can't specify an alternative locale; the system's default locale is used, as in Example 3-11.

Example 3-11. main/src/main/java/strings/Case.java

```
    String name = "Java Cookbook";
    System.out.println("Normal:\t" + name);
    System.out.println("Upper:\t" + name.toUpperCase());
    System.out.println("Lower:\t" + name.toLowerCase());
```

```
String javaName = "java cookBook"; // If it were Java identifiers :-)
if (!name.equals(javaName))
  System.err.println("equals() correctly reports false");
else
  System.err.println("equals() incorrectly reports true");
if (name.equalsIgnoreCase(javaName))
  System.err.println("equalsIgnoreCase() correctly reports true");
else
  System.err.println("equalsIgnoreCase() incorrectly reports false");
```

If you run this, it prints the first name changed to uppercase and lowercase, then it reports that both methods work as expected:

```
C:\javasrc\strings&gt;<strong>java Case.java</strong>
Normal: Java Cookbook
Upper:  JAVA COOKBOOK
Lower:  java cookbook
equals( ) correctly reports false
equalsIgnoreCase( ) correctly reports true
```

Java doesn't provide for uppercasing the first letter of each word (capitalization) of names, but Apache Commons Text does. Add the following dependency to your *pom.xml* or *build.gradle*:

```
org.apache.commons:commons-text:1.12.0
```

The Apache Commons `WordUtils` class provides a number of useful methods worth exploring in its documentation, but Example 3-12 includes examples of the capitalization methods.

Example 3-12. main/src/main/java/strings/Case.java

```
// Fancier stuff via Apache Commons Text:
String smith = "lorry smith";
System.out.println(WordUtils.capitalize(smith));
String jones = "JENkins JONes";
System.out.println(WordUtils.capitalizeFully(jones));
```

This segment prints:

```
Lorry Smith
Jenkins Jones
```

As illustrated, `capitalize()` just ensures the first letter of each word is capitalized, whereas `capitalizeFully()` also lowercases all subsequent letters in each word.

See Also

Regular expressions make it simpler to ignore case in string searching (as shown in Chapter 4).

3.10 Adding Nonprintable Characters into a String

Problem

You need to put nonprintable characters into strings.

Solution

Use the backslash character (`\`) to begin one of the Java string escapes.

Discussion

The Java string escapes are listed in Table 3-4.

Table 3-4. String escapes

To get	Use	Notes
Tab	`\t`	
Linefeed (Unix newline)	`\n`	The call `System.getProperty("line.separator")` will give you the platform's line end.
Carriage return	`\r`	
Form feed	`\f`	
Backspace	`\b`	
Single quote	`\'`	
Double quote	`\"`	
Unicode character	`\u` *NNNN*	Four hexadecimal digits (no `\x` as in C/C++). See the Unicode website for codes.
Octal(!) character	`\` *NNN*	Who uses octal (base 8) these days?
Backslash	`\\`	

Here is a code example that shows most of these in action:

```
public class StringEscapes {
  public static void main(String[] argv) {
    System.out.println("Java Strings in action:");
    System.out.println("A bell (alarm) entered in Octal: \007");
    System.out.println("A tab key: <<<\t>>>");
    System.out.println("A newline: <<<\n>>>");
    System.out.println("A UniCode character: \u0207");
    System.out.println("A backslash character: \\");
  }
}
```

```
A bell (alarm) entered in Octal:
A tab key: <<<  >>>
A newline: <<<
>>>
```

```
A UniCode character: ê
A backslash character: \
```

If you have a lot of non-ASCII characters to enter, you may want to consider using Java's input methods, discussed briefly in the infrequently updated Java 8 online documentation.

Ian's Basic Steps: Internationalization and Localization

Internationalization and localization consist of the following:

Sensitivity training (Internationalization, or I18N)
: Making your software sensitive to these issues

Language lessons (Localization, or L10N)
: Writing configuration files for each language

Culture lessons (optional)
: Customizing the presentation of numbers, fractions, dates, and message formatting

For more information, see *Java Internationalization* by Andy Deitsch and David Czarnecki (O'Reilly).

3.11 Creating a Message to the World with I18N Resources

Problem

You want your program to take sensitivity training so that it can communicate well internationally.

Solution

Use internationalization software to obtain all strings to be displayed.

Discussion

Your program must obtain all control and message strings via the internationalization software. Here's how:

1. Get a `ResourceBundle` that has been saved on disk previously:

   ```
   ResourceBundle rb = ResourceBundle.getBundle("Menus");
   ```

   I'll cover `ResourceBundle` in Recipe 3.13, but briefly, a `ResourceBundle` represents a collection of name-value pairs (resources). The names are names you

assign to each GUI control or other user interface text, and the values are the text to assign to each control in a given language.

2. Use this `ResourceBundle` to fetch the localized version of each control name.

   Old way:

   ```
   String label = "Exit";
   // Create the control, e.g., new JButton(label);
   ```

   New way:

   ```
   try { label = rb.getString("exit.label"); }
   catch (MissingResourceException e) { label="Exit"; } // fallback
   // Create the control, e.g., new JButton(label);
   ```

This may seem like quite a bit of code for one control, but you can write a convenience routine to simplify it, like this:

```
JButton exitButton = I18N.getButton(bundle, "exit.label", "Exit");
```

The file *I18N.java* is included in the book's code distribution (in *darwinsys-api*). There is also *I18NDemo.java* (in *javasrc/main/src/main/java/i18n/I18NDemo.java*).

While the example is a Swing `JButton`, the same approach goes with other UIs, such as the web tier.

What happens at runtime?

The default locale is used because we didn't specify one. The default locale is platform dependent:

Unix/POSIX
: `LANG` environment variable (per user)

Windows
: Control Panel→Regional Settings

macOS
: System Preferences→Language & Text

Others
: See platform documentation

`ResourceBundle.getBundle()` locates a file with the named resource bundle name (`Menus`, in the previous example), plus an underscore and the locale name (if a non-default locale is set), plus another underscore and the locale variation (if any variation is set), plus the extension *.properties*. If a variation is set but the file can't be found, it falls back to just the country code. If that can't be found, it falls back to the original default. Table 3-5 shows some examples for various locales.

Note that Android apps—usually written in Java or Kotlin—use a similar mechanism but with the files in XML format instead of Java Properties and with some small changes in the name of the file in which the properties files are found.

Table 3-5. Properties filenames for different locales

Locale	Filename
Default locale	*Menus.Properties*
Swedish	*Menus_sv.properties*
Spanish	*Menus_es.properties*
Cuban Spanish	*Menus_es_CU.properties*
French	*Menus_fr.properties*
French-Canadian	*Menus_fr_CA.properties*

Locale names are two-letter ISO 639 language codes (lowercase), and they normally abbreviate the country's *endonym* (the name its language speakers refer to it by); thus, Swedish is *sv* for *Sverige*, Spanish is *es* for *Español*, etc. Locale variations are two-letter ISO country codes (uppercase); for example, CA for Canada, US for the United States, SV for Sweden, ES for Spain, etc.

Setting the locale

On Windows, go into Regional Settings in the Control Panel. Changing this setting may entail a reboot, so exit any editor windows.

On Unix, set your `LANG` environment variable. For example, a Korn shell user in Mexico might have this line in their *.profile*:

```
export LANG=es_MX
```

On either system, for testing a different locale, you need only define the locale in the system properties at runtime using the command-line option `-D`, as in:

```
$ java -Duser.language=es Browser.java
```

This runs the Java program named `Browser` in package i18n in the Spanish locale.

You can get a list of the available locales with a call to `Locale.getAvailableLocales()`.

3.12 Using a Particular Locale

Problem

You want to use a locale other than the default in a particular operation.

Solution

Obtain a `Locale` by using a predefined instance or the `Locale` constructor. Optionally make it global to your application by using `Locale.setDefault(newLocale)`.

Discussion

Classes that provide formatting services, such as `DateTimeFormatter` and `NumberFormat`, provide overloads so they can be called either with or without a `Locale`-related argument.

To obtain a `Locale` object, you can employ one of the predefined locale variables provided by the `Locale` class, or you can construct your own `Locale` object giving a language code and a country code:

```
Locale locale1 = Locale.FRANCE;    // predefined
Locale locale2 = new Locale("en", "UK");    // English, UK version
```

These can then be used in the various formatting operations:

```
DateFormat frDateFormatter = DateFormat.getDateInstance(
                DateFormat.MEDIUM, frLocale);
DateFormat ukDateFormatter = DateFormat.getDateInstance(
                DateFormat.MEDIUM, ukLocale);
```

Either of these can be used to format a date or a number, as shown in Example 3-13.

Example 3-13. main/src/main/java/i18n/UseLocales.java

```
public class UseLocales {
  public static void main(String[] args) {

    Locale frLocale = Locale.FRANCE;  // predefined
    Locale ukLocale = Locale.of("en", "UK");  // English, UK version

    DateTimeFormatter defaultDateFormatter =
      DateTimeFormatter.ofLocalizedDateTime(
        FormatStyle.MEDIUM);
    DateTimeFormatter frDateFormatter =
      DateTimeFormatter.ofLocalizedDateTime(
        FormatStyle.MEDIUM).localizedBy(frLocale);
    DateTimeFormatter ukDateFormatter =
      DateTimeFormatter.ofLocalizedDateTime(
        FormatStyle.MEDIUM).localizedBy(ukLocale);

    LocalDateTime now = LocalDateTime.now();
    System.out.println("Default: " + ' ' +
      now.format(defaultDateFormatter));
    System.out.println(frLocale.getDisplayName() + ' ' +
      now.format(frDateFormatter));
    System.out.println(ukLocale.getDisplayName() + ' ' +
```

```
            now.format(ukDateFormatter));
    }
}
```

The program prints the locale name and formats the date in each of the locales:

```
$ java UseLocales.java
Default:  Oct 16, 2025, 4:41:45 PM
French (France) 16 oct. 2025 à 16:41:45
English (UK) Oct 16, 2025, 4:41:45 PM
$
```

3.13 Creating a Resource Bundle

Problem

You need to create a resource bundle for use with I18N.

Solution

A resource bundle is simply a collection of names and values. You could write a `java.util.ResourceBundle` subclass, but it is easier to create textual `Properties` files (see Recipe 7.10) that you then load with `ResourceBundle.getBundle()`. The files can be created using any plain-text editor. Leaving it in a text file format also allows user customization in desktop applications; a user whose language is not provided for, or who wishes to change the wording somewhat due to local variations in dialect, should be able to edit the file.

Note that the resource bundle text file should not have the same name as any of your Java classes. The reason for this is that the `ResourceBundle` constructs a class dynamically with the same name as the resource files.

Discussion

Here is a sample properties file for a few menu items:

```
# Default Menu properties
# The File Menu
file.label=File Menu
file.new.label=New File
file.new.key=N
file.save.label=Save
file.new.key=S
```

Creating the default properties file is usually not a problem, but creating properties files for other languages might be. Unless you are a large multinational corporation, you will probably not have the resources (pardon the pun) to create resource files in-house. If you are shipping commercial software or using the web for global reach, you

need to identify your target markets and understand which of these are most sensitive to wanting menus and the like in their own languages. Then, hire a professional translation service that has expertise in the required languages to prepare the files. Test them well—using native speakers of the target language, not affiliated with the translation house—before you ship, as you would any other part of your software.

If you need special characters, multiline text, or other complex entry, remember that a `ResourceBundle` is also a `Properties` file, so see the documentation for `java.util.Properties`.

3.14 Program: A Simple Text Formatter

This program is a primitive text formatter, representative of what people used on most computing platforms before the rise of standalone graphics-based word processors, laser printers, and, eventually, desktop publishing and office suites. It simply reads words from a file, previously created with a text editor, and outputs them until it reaches the right margin, when it calls `println()` to append a line ending. For example, here is an input file:

```
It's a nice
day, isn't it, Mr. Mxyzzptllxy?
I think we should go for a walk.
```

Given that file as the input, the `Fmt` program prints the lines formatted neatly:

```
It's a nice day, isn't it, Mr. Mxyzzptllxy? I think we should go for a walk.
```

As you can see, it fits the text we gave it to the margin and discards all single line breaks present in the original. The code is in the *javasrc* repo at *main/src/main/java/strings/Fmt.java* and online at my GitHub repository.

A slightly fancier version of this program, `Fmt2`, is also available at *include main/src/main/java/strings/Fmt2.java* or online at my GitHub repository.

The `Fmt2` version uses *dot commands*—lines beginning with periods—to give limited control over the formatting. A family of dot-command formatters includes Unix's `roff`, `nroff`, `troff`, and `groff`, which are in the same family with programs called `runoff` on Digital Equipment systems. The original program for this is believed to be J. Saltzer's `runoff`, which first appeared on Multics and from there made its way into various OSes. Incidentally, `Fmt2` subclasses `Fmt` and just overrides the `format()` method to include additional functionality.

CHAPTER 4

String Matching with Regular Expressions

4.0 Introduction

Suppose you have been on the internet for a few years and have been faithful about saving all your correspondence, just in case you (or your lawyers, or the prosecution) need a copy. The result is that you have a 5 GB disk partition dedicated to saved mail. Let's further suppose that you remember that somewhere in there is an email message from someone named Angie or Anjie. Or was it Angy? But you don't remember what you called it or where you stored it. Obviously, you have to look for it.

But while some of you might go and try to open up all 15 million documents in a word processor, I'll just find it with one simple command. Any system that provides regular expression support allows me to search for the pattern in several ways. An easy one to understand is:

```
Angie|Anjie|Angy
```

which you can probably guess means just to search for any one of the variations. A more concise form (more thinking, less typing) is:

```
An[^ dn]
```

The syntax will become clear as we go through this chapter. Briefly, the "A" and the "n" match themselves, in effect finding words that begin with "An." The cryptic `[^ dn]` requires the "An" to be followed by a character other than (`^` means *not* in this context) a space (to eliminate the very common English word "an" at the start of a sentence), "d" (to eliminate the common word "and") or "n" (to eliminate "Anne," "Announcing," etc.).

Has your word processor gotten past its splash screen yet? Well, it doesn't matter, because I've already found the missing file. To find the answer, I just typed this command (it'll work on any Unix/Linux/macOS system):

```
grep 'An[^ dn]' *
```

Regular expressions, or *regexes* for short, provide a concise and precise specification of patterns to be matched in text. One good way to think of regular expressions is as a little language for matching patterns of characters in text contained in strings. A regular expression API is an interpreter for matching regular expressions.

As another example of the power of regular expressions, consider the problem of bulk-updating hundreds of files. When I started with Java, the syntax for declaring array references was `baseType arrayVariableName[]`. For example, a method with an array argument, such as every program's `main` method, was commonly written as:

```
public static void main(String args[]) {
```

But as time went by, it became clear to the stewards of the Java language that it would be better to write it as `baseType[] arrayVariableName`, like this:

```
public static void main(String[] args) {
```

This is better Java style because it associates the "array-ness" of the type with the type itself, rather than with the local argument name. While Java still accepts the old form, there is a strong preference for the new syntax.[1]

So I wanted to change all occurrences of `main` written the old way to the new way. I used the pattern `main(String [a-z]` with the `grep` utility described earlier to find the names of all the files containing old-style main declarations (i.e., `main(String` followed by a space and a name character rather than an open square bracket). I then used another regex-based Unix tool, the stream editor `sed`, in a little shell script to change all occurrences in those files from `main(String ([a-z][a-z])[]` to `main(String[] $1` (the regex syntax used here is discussed later in this chapter). Again, the regex-based approach was orders of magnitude faster than doing it interactively, even using a reasonably powerful editor such as `vi` or `Emacs`, let alone trying to use a graphical word processor.

Historically, the syntax of regexes has changed as they get incorporated into more tools and more languages, so the exact syntax in the previous examples is not exactly

1 We're starting to see contexts where the old form isn't accepted (e.g., as a field in a classless main).

what you'd use in Java, but it does convey the conciseness and power of the regex mechanism.[2]

As a third example, consider parsing an Apache web server logfile, where some fields are delimited with quotes, others with square brackets, and others with spaces. Writing ad hoc code to parse this is messy in any language, but a well-crafted regex can break the line into all its constituent fields in one operation (this example is developed in Example 4-3).

These same time gains can be had by Java developers. Regular expression support has been in the standard Java runtime for ages and is well integrated (e.g., there are regex methods in the standard class `java.lang.String` and in the no-longer-new "new I/O" package java.nio). There are a few other regex packages for Java, and you may occasionally encounter code using them, but most code from this century can be expected to use the built-in package. The syntax of Java regexes themselves is discussed in Recipe 4.1, and the syntax of the Java API for using regexes is described in Recipe 4.2. The remaining recipes show some applications of regex technology in Java.

See Also

Mastering Regular Expressions by Jeffrey Friedl (O'Reilly) is the definitive guide to all the details of regular expressions. Most introductory books on Unix and Perl include some discussion of regexes; *Unix Power Tools* by Jerry Peek, Shelley Powers, Tim O'Reilly, and Mike Loukides (O'Reilly) devotes a chapter to them.

4.1 Regular Expression Syntax

Problem

You need to learn the syntax of Java regular expressions.

Solution

Consult Table 4-1 for a list of the regular expression characters.

2 Non-Unix fans fear not, for you can use tools like `grep` on Windows systems using one of several packages. One is an open source package called git bash, which includes `git` and several tools including `grep`. Another is Microsoft's `findstr /R` command for Windows. Or you can use my JGrep program in Recipe 4.10 if you don't have `grep` on your system. Incidentally, the name `grep` comes from an ancient Unix line editor command `g/RE/p`, the command to find the regex globally in all lines in the edit buffer and print the lines that match—just what the `grep` program does to lines in files.

Discussion

These pattern characters let you specify regexes of considerable power. In building patterns, you can use any combination of ordinary text and the *metacharacters*, or special characters, in Table 4-1. These can all be used in any combination that makes sense. For example, `a+` means any number of occurrences of the letter `a`, from one up to a million or a gazillion. The pattern `Mrs?\.` matches `Mr.` or `Mrs.` And `.*` indicates any character, any number of times, and is similar in meaning to most command-line interpreters' meaning of the `*` alone. The pattern `\d+` means any number of numeric digits. `\d{2,3}` means a two- or three-digit number.

Table 4-1. Regular expression metacharacter syntax

Subexpression	Matches	Notes
General		
`^`	Start of line/string	
`$`	End of line/string	
`\b`	Word boundary	
`\B`	Not a word boundary	
`\A`	Beginning of entire string	
`\z`	End of entire string	
`\Z`	End of entire string (except allowable final line terminator)	See Recipe 4.9
`.`	Any one character (except line terminator)	
`[...]`	"Character class"; any one character from those listed	
`[^...]`	Any one character not from those listed	See Recipe 4.2
Alternation and grouping		
`(...)`	Grouping (capture groups)	See Recipe 4.4
`|`	Alternation	
`(?:_re_)`	Noncapturing parenthesis	
`\G`	End of the previous match	
`\_n`	Back-reference to capture group number *n*	
Normal (greedy) quantifiers		
`{` *m*`,`*n*`}`	Quantifier for from *m* to *n* repetitions	See Recipe 4.5
`{` *m* `,}`	Quantifier for *m* or more repetitions	
`{` *m* `}`	Quantifier for exactly *m* repetitions	See Recipe 4.9
`{,`*n*`}`	Quantifier for 0 up to *n* repetitions	
`*`	Quantifier for 0 or more repetitions	Short for `{0,}`

Subexpression	Matches	Notes
`+`	Quantifier for 1 or more repetitions	Short for `{1,}`; see Recipe 4.2
`?`	Quantifier for 0 or 1 repetitions (i.e., present exactly once, or not at all)	Short for `{0,1}`
Reluctant (nongreedy) quantifiers		
`{` *m*`,`*n*`}?`	Reluctant quantifier for from *m* to *n* repetitions	
`{` *m* `,}?`	Reluctant quantifier for *m* or more repetitions	
`{,`*n*`}?`	Reluctant quantifier for 0 up to *n* repetitions	
`*?`	Reluctant quantifier: 0 or more	
`+?`	Reluctant quantifier: 1 or more	See Recipe 4.9
`??`	Reluctant quantifier: 0 or 1 times	
Possessive (very greedy) quantifiers		
`{` *m*`,`*n*`}+`	Possessive quantifier for from *m* to *n* repetitions	
`{` *m* `,}+`	Possessive quantifier for *m* or more repetitions	
`{,`*n*`}+`	Possessive quantifier for 0 up to *n* repetitions	
`*+`	Possessive quantifier: 0 or more	
`++`	Possessive quantifier: 1 or more	
`?+`	Possessive quantifier: 0 or 1 times	
Escapes and shorthands		
`\`	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic characters into metacharacters	
`\Q`	Escape (quote) all characters up to `\E`	
`\E`	Ends quoting begun with `\Q`	
`\t`	Tab character	
`\r`	Return (carriage return) character	
`\n`	Newline character	See Recipe 4.9
`\f`	Form feed	
`\w`	Character in a word	Use `\w+` for a word; see Recipe 4.9
`\W`	A nonword character	
`\d`	Numeric digit	Use `\d+` for an integer; see Recipe 4.2
`\D`	A nondigit character	
`\s`	Whitespace	Space, tab, etc., as determined by `java.lang.Character.isWhitespace()`
`\S`	A nonwhitespace character	See Recipe 4.9

Subexpression	Matches	Notes
Unicode blocks (representative samples)		
`\p{InGreek}`	A character in the Greek block	(Simple block)
`\P{InGreek}`	Any character not in the Greek block	
`\p{Lu}`	An uppercase letter	(Simple category)
`\p{Sc}`	A currency symbol	
POSIX-style character classes (defined only for US-ASCII)		
`\p{Alnum}`	Alphanumeric characters	`[A-Za-z0-9]`
`\p{Alpha}`	Alphabetic characters	`[A-Za-z]`
`\p{ASCII}`	Any ASCII character	`[\x00-\x7F]`
`\p{Blank}`	Space and tab characters	
`\p{Space}`	Space characters	`[ \t\n\x0B\f\r]`
`\p{Cntrl}`	Control characters	`[\x00-\x1F\x7F]`
`\p{Digit}`	Numeric digit characters	`[0-9]`
`\p{Graph}`	Printable and visible characters (not spaces or control characters)	
`\p{Print}`	Printable characters	Same as `\p{Graph}`
`\p{Punct}`	Punctuation characters	One of `!"#$%&'()\*+,-./:;<=>?@[]\^_`{|}\~`
`\p{Lower}`	Lowercase characters	`[a-z]`
`\p{Upper}`	Uppercase characters	`[A-Z]`
`\p{XDigit}`	Hexadecimal digit characters	`[0-9a-fA-F]`

Regexes match any place possible in the string. Patterns followed by greedy quantifiers (the only type that existed in traditional Unix regexes) consume (match) as much as possible without compromising any subexpressions that follow. Patterns followed by possessive quantifiers match as much as possible without regard to following subexpressions. Patterns followed by reluctant quantifiers consume as few characters as possible to still get a match.

Also, unlike regex packages in some other languages, the Java regex package was designed to handle Unicode characters from the beginning. The standard Java escape sequence `\u+nnnn` is used to specify a Unicode character in the pattern. We use methods of `java.lang.Character` to determine Unicode character properties, such as whether a given character is a space.

To teach students how regexes work, I provide a little program called REDemo.[3] The code for REDemo is too long to include in the book; in the online directory *regex* of the *darwinsys-api* repo, you will find *REDemo.java*, which can be run to explore how regexes work. It's also available online at my GitHub repository.

In the uppermost text box (see Figure 4-1), type the regex pattern you want to test. As you type each character, the regex is checked for syntax; if the syntax is OK, you see a checkmark beside it. You can then select Match, Find, or Find All. Match means that the entire string must match the regex, and Find means the regex must be found somewhere in the string. (Find All counts the number of occurrences that are found.) Below that, you type a string that the regex is to match against. Experiment to your heart's content. When you have the regex the way you want it, you can paste it into your Java program. You'll need to escape (backslash) any characters that are treated specially by both the Java compiler and the Java regex package, such as the backslash itself, double quotes, and others. Once you get a regex the way you want it, there is a Copy button (not shown in these screenshots) to export the regex to the clipboard, with or without backslash doubling, depending on how you want to use it.

Remember that because a regex is entered as a string that will be compiled by a Java compiler, you usually need two levels of escaping for any special characters, including backslash and double quotes. For example, the regex (which includes the double quotes):

```
"You said it\."
```

has to be typed like this to be a valid compile-time Java language `String`:

```
String pattern = "\"You said it\\.\""
```

In Java 14+ you can also use a `String` text block to avoid escaping the quotes:

```
String pattern = """
        "You said it\\."""
```

I can't tell you how many times I've made the mistake of forgetting the extra backslash in `\d+`, `\w+`, and their kin!

In Figure 4-1, I typed `qu` into the REDemo program's Pattern box, which is a syntactically valid regex pattern: any ordinary characters stand as regexes for themselves, so this looks for the letter `q` followed by `u`. In the top version, I typed only a `q` into the string, and this is not matched. In the second, I have typed `quack` and the `q` of a sec-

3 REDemo was inspired by (but does not use any code from) a similar program provided with the now-retired Apache Jakarta Regular Expressions package.

ond quack. Because I have selected Find All, the count shows one match. As soon as I type the second u, the count is updated to two, as shown in the third version.

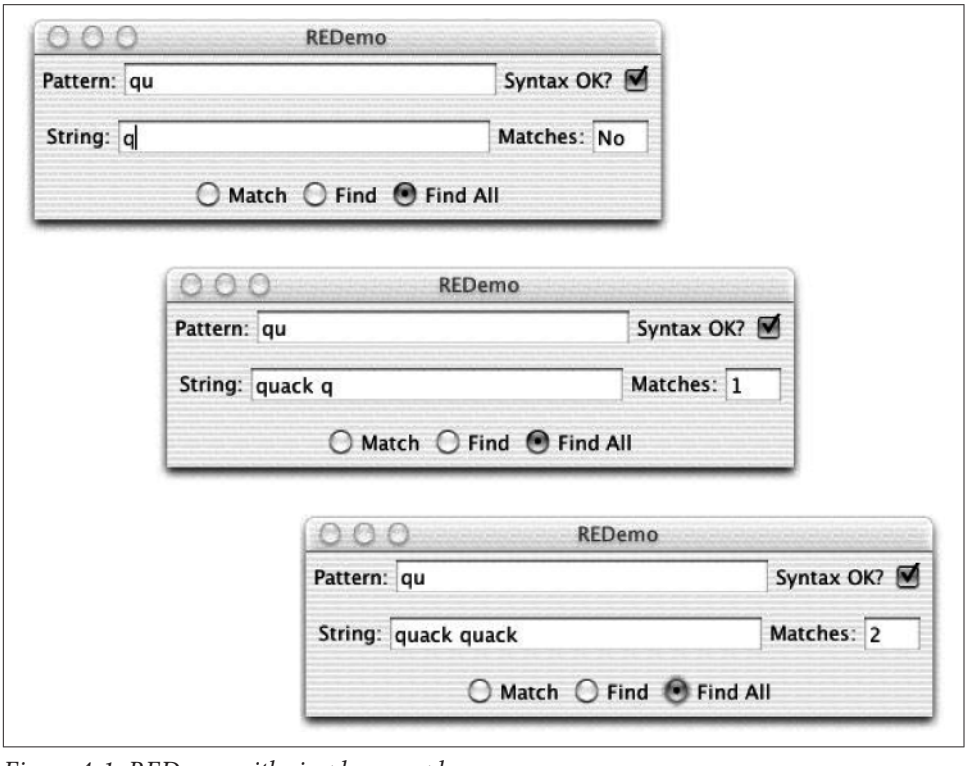


Figure 4-1. REDemo with simple examples

Regexes can do far more than just character matching. For example, the two-character regex `^T` would match beginning of line (`^`) immediately followed by a capital T—that is, any line beginning with a capital T. It doesn't matter whether the line begins with "Tiny trumpets," "Titanic tubas," or "Triumphant twisted trombones," as long as the capital T is present in the first position.

But here we're not very far ahead. Have we really invested all this effort in regex technology just to be able to do what we could already do with the `java.lang.String` method `startsWith()`? Hmmm, I can hear some of you getting a bit restless. Stay in your seats! What if you wanted to match not only a letter T in the first position, but also a vowel immediately after it, followed by any number of letters in a word, followed by an exclamation point? Surely you could do this in Java by checking `startsWith("T")` and `charAt(1) == 'a' || charAt(1) == 'e'`, and so on? Yes, but by the time you did that, you'd have written a lot of very highly specialized code that you couldn't use in any other application. With regular expressions, you can just give the pattern `^T[aeiou]\w*!`. That is, `^` and `T` as before, followed by a character class listing

the vowels, followed by any number of word characters (`\w*`), followed by the exclamation point.

"But wait, there's more!" as my late, great boss Yuri Rubinsky used to say. What if you want to be able to change the pattern you're looking for *at runtime*? Remember all that Java code you just wrote to match `T` in column 1, plus a vowel, some word characters, and an exclamation point? Well, it's time to throw it out. Because this morning we need to match `Q`, followed by a letter other than `u`, followed by a number of digits, followed by a period. While some of you start writing a new function to do that, the rest of us will just saunter over to the RegEx Bar & Grille, order a `^Q[^u]\d+\.` from the bartender, and be on our way.

OK, if you want an explanation: the `[^u]` means match any one character that is not the character `u`. The `\d+` means one or more numeric digits. The `+` is a quantifier meaning one or more occurrences of what it follows, and `\d` is any one numeric digit. So `\d+` means a number with one, two, or more digits. Finally, the `\.`? Well, `.` by itself is a metacharacter. Most single metacharacters are switched off by preceding them with an escape character. Not the Esc key on your keyboard, of course. The regex escape character is the backslash. Preceding a metacharacter like `.` with this escape turns off its special meaning, so we look for a literal period rather than any character. Preceding a few selected alphabetic characters (e.g., `n`, `r`, `t`, `s`, `w`) with escape turns them into metacharacters.

Figure 4-2 shows the `^Q[^u]\d+\.` regex in action. In the first frame, I have typed part of the regex as `^Q[^u`. Because there is an unclosed square bracket, the Syntax OK flag is turned off; when I complete the regex, it will be turned back on. In the second frame, I have finished typing the regex, and I've typed the data string as `QA577` (which you should expect to match the `^Q[^u]\\d+` but not the period since I haven't typed it). In the third frame, I've typed the period, so the Matches flag is set to Yes.

Because backslashes need to be escaped when pasting the regex into Java code, the current version of REDemo has both a Copy Pattern button, which copies the regex verbatim for use in documentation and in Unix commands, and a Copy Pattern Backslashed button, which copies the regex to the clipboard with backslashes doubled, for pasting into Java strings.

By now you should have at least a basic grasp of how regexes work in practice. The rest of this chapter gives more examples and explains some of the more powerful topics, such as capture groups. As for how regexes work in theory—and there are a lot of theoretical details and differences among regex flavors—the interested reader is referred to *Mastering Regular Expressions*. Meanwhile, let's start learning how to write Java programs that use regular expressions.

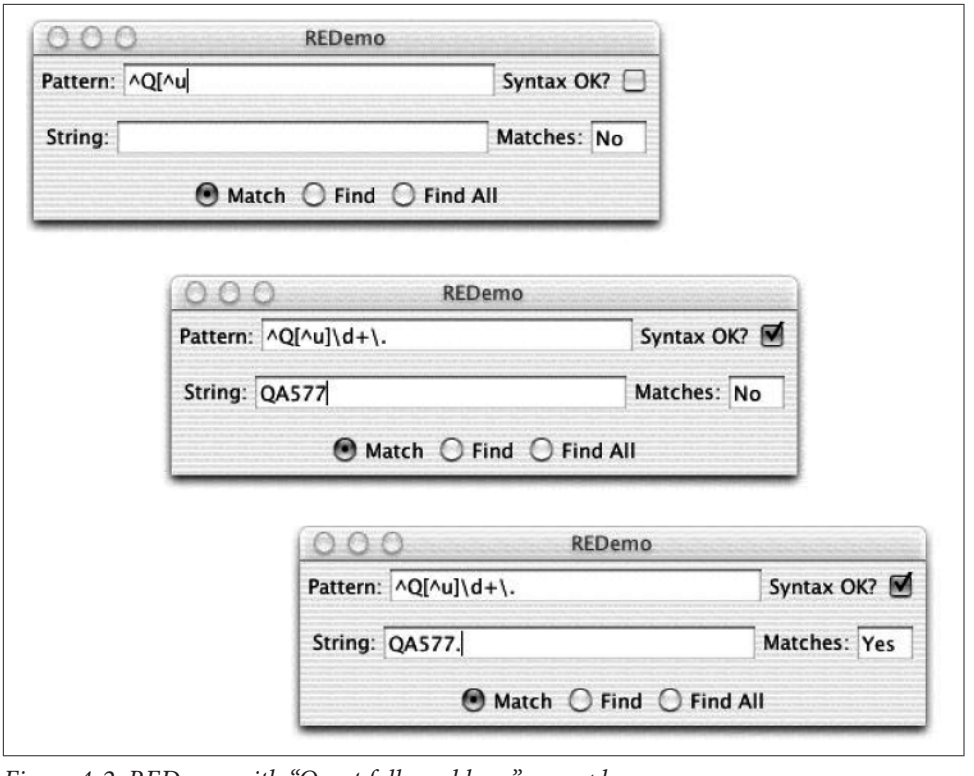


Figure 4-2. REDemo with "Q not followed by u" example

4.2 Checking If a String Matches a Regex

Problem

You're ready to get started using regular expression processing to beef up your Java code by testing to see if a given pattern can match in a given string.

Solution

Use the Java regular expressions package, java.util.regex.

Discussion

The good news is that the Java API for regexes is actually easy to use. If all you need is to find out whether a given regex matches a string, you can use the convenient `boolean matches()` method of the `String` class, which accepts a regex pattern in `String` form as its argument:

```
    if (inputString.matches(stringRegexPattern)) {
        // it matched... do something with it...
    }
```

This is, however, a convenience routine, and convenience always comes at a price. If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a `Pattern` and its `Matcher`(s). A complete program constructing a `Pattern` and using it to match against strings is shown in Example 4-1.

Example 4-1. main/src/main/java/regex/RESimple.java

```
public class RESimple {
  public static void main(String[] argv) {
    String pattern = "^Q[^u]\\d+\\.";
    String[] input = {
      "QA777. is the next flight. It is on time.",
      "Quack, Quack, Quack!"
    };

    Pattern p = Pattern.compile(pattern);

    for (String in : input) {
      boolean found = p.matcher(in).lookingAt();

      System.out.println("'" + pattern + "'" +
      (found ? " matches '" : " doesn't match '") + in + "'");
    }
  }
}
```

The java.util.regex package contains two classes, `Pattern` and `Matcher`, which provide the public API shown in Example 4-2.

Example 4-2. Regex public API

```
/**
 * The main public API of the java.util.regex package.
 */
package java.util.regex;

public final class Pattern {
  // Flags values ('or' together)
  public static final int
    UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE,
    DOTALL, UNICODE_CASE, CANON_EQ;
  // No public constructors; use these Factory methods
  public static Pattern compile(String patt);
  public static Pattern compile(String patt, int flags);
  // Method to get a Matcher for this Pattern
  public Matcher matcher(CharSequence input);
```

```
  // Information methods
  public String pattern();
  public int flags();
  // Convenience methods
  public static boolean matches(String pattern, CharSequence input);
  public String[] split(CharSequence input);
  public String[] split(CharSequence input, int max);
}

public final class Matcher {
  // Action: find or match methods
  public boolean matches();
  public boolean find();
  public boolean find(int start);
  public boolean lookingAt();
  // "Information about the previous match" methods
  public int start();
  public int start(int whichGroup);
  public int end();
  public int end(int whichGroup);
  public int groupCount();
  public String group();
  public String group(int whichGroup);
  // Reset methods
  public Matcher reset();
  public Matcher reset(CharSequence newInput);
  // Replacement methods
  public Matcher appendReplacement(StringBuffer where, String newText);
  public StringBuffer appendTail(StringBuffer where);
  public String replaceAll(String newText);
  public String replaceFirst(String newText);
  // information methods
  public Pattern pattern();
}

/* String, showing only the RE-related methods */
public final class java.lang.String {
  public boolean matches(String regex);
  public String replaceFirst(String regex, String newStr);
  public String replaceAll(String regex, String newStr);
  public String[] split(String regex);
  public String[] split(String regex, int max);
  ...
}
```

This API is large enough to require some explanation. These are the typical steps for regex matching in a production program:

1. Create a `Pattern` by calling the static method `Pattern.compile()`.
2. Request a `Matcher` from the `Pattern` by calling `pattern.matcher(CharSequence)` for each `String` (or other `CharSequence`) you wish to look through.

3. Call (once or more) one of the finder methods (discussed later in this section) in the resulting `Matcher`.

The `java.lang.CharSequence` interface provides simple read-only access to objects containing a collection of characters. The standard implementations are `String` and `StringBuffer`/`StringBuilder` (described in Chapter 3), and the new I/O class `java.nio.CharBuffer`.

Of course, you can perform regex matching in other ways, such as using the convenience methods in `Pattern` or even in `java.lang.String`, like this:

```
public class StringConvenience {
  public static void main(String[] argv) {

    String pattern = ".*Q[^u]\\d+\\..*";
    String line = "Order QT300. Now!";
    if (line.matches(pattern)) {
      System.out.println(line + " matches \"" + pattern + "\"");
    } else {
      System.out.println("NO MATCH");
    }
  }
}
```

But the three-step list is the standard pattern for matching. You'd likely use the `String` convenience routine in a program that only used the regex once; if the regex were being used more than once, it is worth taking the time to compile it because the compiled version runs faster.

In addition, the `Matcher` has several finder methods, which provide more flexibility than the `String` convenience routine `match()`. These are the `Matcher` methods:

`match()`
: Used to compare the entire string against the pattern; this is the same as the routine in `java.lang.String`. Because it matches the entire `String`, I had to put `.*` before and after the pattern.

`lookingAt()`
: Used to match the pattern only at the beginning of the string.

`find()`
: Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Each of these methods returns `boolean`, with `true` meaning a match and `false` meaning no match. To check whether a given string matches a given pattern, you need only type something like the following:

```
Matcher m = Pattern.compile(patt).matcher(line);
if (m.find( )) {
    System.out.println(line + " matches " + patt)
}
```

When constructing a `Pattern` whose regex syntax is complex, you can use the multiline or commented style by passing `Pattern.COMMENTS` as the second argument to the `compile()` method:

```
Pattern.compile("""
        \s*     # leading space
        \d+     # number
        \w+     # name
""", Pattern.COMMENTS);
```

It's recommended to use the `String` text block style as shown (discussed near the end of the introduction to Chapter 3), and it's important to *not* put commas after each component. A longer example is shown in the following recipe, Recipe 4.3.

You may want more flexibility to extract the part of text that matched. This is the subject of the next recipes, which cover uses of the Matcher API. Initially, the examples just use arguments of type `String` as the input source. Use of other `CharSequence` types is covered in Recipe 4.6.

4.3 Grouping: Specifying Parts of the Regex

Problem

You need to refer to text that matched part of the regex, rather than the text that matched the entire regex.

Solution

Use parenthesized groups to delimit the subpart of the regex. Use the `Matcher` method `group()` to refer to the corresponding submatch.

Discussion

Using groups to enable access to part of a regex is a fundamental part of regex work. Suppose we need to parse text like a logfile, which has a number of fields or columns. We want to print out individual fields from the text. Example 4-3 demonstrates how to do this.

Example 4-3. main/src/main/java/regex/LogRegEx.java—Apache Log File Scanner

```
public class LogRegEx {

  public static final int MIN_FIELDS = 8;

  final static String SAMPLE_LINE = "123.45.67.89 - - [27/Oct/2000:09:27:09 -0400]
\"GET /java/javaResources.html HTTP/1.0\" 200 10450 \"-\" \"Mozilla/4.6 [en] (X11;
U; OpenBSD 2.8 i386; Navigator)\"";

  final static String LOG_ENTRY_PATTERN = """
    ^([\\w\\d.-]+)\\s+                # 1 - IP
    (\\S+)\\s+                        # 2 - User, from identd (always "-")
    (\\S+)\\s+                        # 3 - User, from https (often "-")
    \\[([\\w:/]+\\s[+-]\\d{4})\\]\\s+ # 4 - Date, time, space, timezone-offset
    ([a-zA-Z.]+\\s+)?                 # 5 - domainname, in some formats
    "(.+?)"\\s+                       # 6 - Entire request line
    (\\d{3})\\s+                      # 7 - Status code (200, 404, etc)
    (\\d+)\\s*                        # 8 - Byte count
    ("[^"]+"\\s*)?                    # 9 - Referrer, or "-"
    ("([^"]+)")?                      # 10 - Browser advertising clause, free-form
    """;

  final static Pattern PATT = Pattern.compile(LOG_ENTRY_PATTERN, Pattern.COMMENTS);

  public static void main(String argv[]) throws IOException {

    System.out.println("RE Pattern:");
    System.out.println(PATT);

    if (argv.length == 0) {
      process(SAMPLE_LINE);
    } else {
      for (String fileName : argv) {
        Files.lines(Path.of(fileName)).forEach(LogRegEx::process);
      }
    }
  }

  static void process(String logEntryLine) {

    System.out.println("Input line:" + logEntryLine);
    Matcher matcher = PATT.matcher(logEntryLine);
    if (!matcher.find()) {
      System.err.println("Failed to match (Bad log entry or problem with regex)");
      return;
    }
    if (matcher.groupCount() < MIN_FIELDS) {
      System.err.println("Matched, but has too few fields):");
      return;
    }
    System.out.println("IP Address: " + matcher.group(1));
```

```
    System.out.println("UserName: " + matcher.group(3));
    System.out.println("Date/Time: " + matcher.group(4));
    System.out.println("Request: " + matcher.group(5));
    System.out.println("Response: " + matcher.group(7));
    System.out.println("Byte Count: " + matcher.group(8));
    if (!matcher.group(9).equals("-"))
      System.out.println("Referer: " + matcher.group(9));
    System.out.println("User-Agent: " + matcher.group(10));
  }
}
```

4.4 Finding the Matching Text

Problem

You need to find the text that the regex matched.

Solution

Use the methods of the `Matcher` class.

Discussion

Sometimes you need to know more than just whether a regex matched a string. In editors and many other tools, you want to know exactly which characters were matched. Remember that with quantifiers such as *, the length of the text that was matched may have no relationship to the length of the pattern that matched it. Do not underestimate the mighty .*, which happily matches thousands or millions of characters if allowed to. As you saw in the previous recipe, you can find out whether a given match succeeds just by using `find()` or `matches()`. But in other applications, you will want to get the characters that the pattern matched.

After a successful call to one of the preceding methods, you can use these information methods on the `Matcher` to get information on the match:

`start(), end()`
: Returns the character position in the string of the starting and ending characters that matched.

`groupCount()`
: Returns the number of parenthesized capture groups, if any; returns 0 if no groups were used.

`group(int i)`
: Returns the characters matched by group *`i`* of the current match, if *`i`* is greater than or equal to zero and less than or equal to the return value of `groupCount()`. Group 0 is the entire match, so `group(0)` (or just `group()`) returns the entire portion of the input that matched.

The notion of parentheses, or capture groups, is central to regex processing. Regexes may be nested to any level of complexity. The `group(int)` method lets you retrieve the characters that matched a given parenthesis group. If you haven't used any explicit parens, you can just treat whatever matched as level zero. Example 4-4 shows part of *REMatch.java*.

Example 4-4. Part of main/src/main/java/regex/REMatch.java

```
public class REmatch {
  public static void main(String[] argv) {

    String patt = "Q[^u]\\d+\\.";
    Pattern r = Pattern.compile(patt);
    String line = "Order QT300. Now!";
    Matcher m = r.matcher(line);
    if (m.find()) {
      System.out.println(patt + " matches \"" +
        m.group(0) +
        "\" in \"" + line + "\"");
    } else {
      System.out.println("NO MATCH");
    }
  }
}
```

When run, this prints:

```
Q[^u]\d+\. matches "QT300." in "Order QT300. Now!"
```

With the `Match` button checked, REDemo provides a display of all the capture groups in a given regex; one example is shown in Figure 4-3.

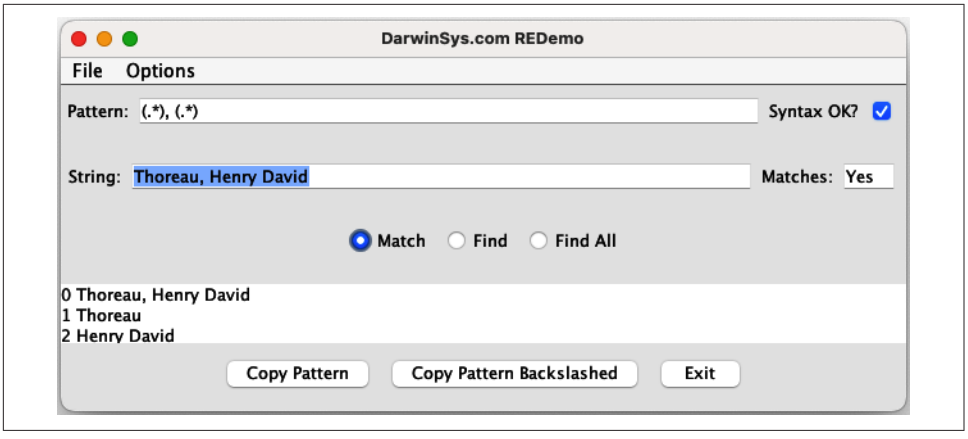


Figure 4-3. REDemo in action

It is also possible to get the starting and ending indices and the length of the text that the pattern matched (remember that terms with quantifiers, such as the `\d+` in this example, can match an arbitrary number of characters in the string). You can use these in conjunction with the `String.substring()` methods as follows:

```
String patt = "Q[^u]\\d+\\.";
Pattern r = Pattern.compile(patt);
String line = "Order QT300. Now!";
Matcher m = r.matcher(line);
if (m.find()) {
  System.out.println(patt + " matches \"" +
    line.substring(m.start(0), m.end(0)) +
    "\" in \"" + line + "\"");
} else {
  System.out.println("NO MATCH");
}
```

Suppose you need to extract several items from a string. If the input is:

```
Smith, John
Adams, John Quincy
```

and you want to get out:

```
John Smith
John Quincy Adams
```

just use the code in Example 4-5.

Example 4-5. main/src/main/java/regex/REmatchTwoFields.java

```
public class REmatchTwoFields {
  public static void main(String[] args) {
    String inputLine = "Adams, John Quincy";
```

```
    // Construct an RE with parens to "grab" last name and first name
    Pattern p = Pattern.compile("(.*), (.*)");
    Matcher m = p.matcher(inputLine);
    if (!m.matches()) {
      throw new IllegalArgumentException("Bad input");
    }
    System.out.println("Numbered: " + m.group(2) + ' ' + m.group(1));

    // Same thing but with names:
    Pattern p2 = Pattern.compile("(?<last>.*), (?<first>.*)");
    m = p2.matcher(inputLine);
    if (!m.matches()) {
      throw new IllegalArgumentException("Bad input");
    }
    System.out.println("Named 1:  " + m.group("first") + " " + m.group("last"));
    System.out.println("Named 2:  " + m.replaceAll("${first} ${last}"));
  }
}
```

While numbered groups are fine when there's only a few, as the regex grows more complex, it makes sense to use named groups, since adding or removing one renumbers all those after it. While numbered groups are entered using `(subPattern)`, named groups are entered using `(?<_name_>subPattern)`, as in the second part of Example 4-5. These can be used in calls to `matcher.group("`*name*`")` or using `${`*name*`}` in the `matcher.replace...` methods.

4.5 Replacing the Matched Text

Problem

Having found some text using a `Pattern`, you want to replace the text with different text, without disturbing the rest of the string.

Solution

As we saw in the previous recipe, regex patterns involving quantifiers can match a lot of characters with very few metacharacters. We need a way to replace the text that the regex matched without changing other text before or after it. We could do this manually using the `String` method `substring()`. However, because it's such a common requirement, the Java Regular Expression API provides some substitution methods.

Discussion

The `Matcher` class provides several methods for replacing just the text that matched the pattern. In all these methods, you pass in the replacement text, or "righthand side" of the substitution. (This term is historical: in a command-line text editor's substitute

command, the lefthand side is the pattern and the righthand side is the replacement text.) These are the replacement methods:

`replaceAll(newString)`
: Replaces all occurrences that matched with the new string

`replaceFirst(newString)`
: Similar to `replaceAll(newString)` but only replaces the first occurence

`appendReplacement(StringBuffer, newString)`
: Copies up to before the first match, plus the given `newString`

`appendTail(StringBuffer)`
: Appends text after the last match (normally used after `appendReplacement`)

Despite their names, the `replace*` methods behave in accordance with the immutability of `Strings` (see "Timeless, Immutable, and Unchangeable" on page 111): they create a new `String` object with the replacement performed. They do not (indeed, could not) modify the string referred to in the `Matcher` object.

Example 4-6 shows use of these three methods.

Example 4-6. main/src/main/java/regex/ReplaceDemo.java

```
/**
 * Quick demo of RE substitution: correct U.S. 'favor'
 * to Canadian/British 'favour', but not in "favorite"
 */
public class ReplaceDemo {
  public static void main(String[] argv) {

    // Make an RE pattern to match as a word only (\b=word boundary)
    String patt = "\\bfavor\\b";

    // A test input.
    String input = "Do me a favor? Fetch my favorite.";
    System.out.println("Input: " + input);

    // Run it from an RE instance and see that it works
    Pattern r = Pattern.compile(patt);
    Matcher m = r.matcher(input);
    System.out.println("ReplaceAll: " + m.replaceAll("favour"));

    // Show the appendReplacement method
    m.reset();
    StringBuilder sb = new StringBuilder();
    System.out.print("Append methods: ");
    while (m.find()) {
      // Copy to before first match,
      // plus the word "favor"
```

```
      m.appendReplacement(sb, "favour");
    }
    m.appendTail(sb);    // copy remainder
    System.out.println(sb.toString());
  }
}
```

Sure enough, when you run it, it does what we expect:

```
Input: Do me a favor? Fetch my favorite.
ReplaceAll: Do me a favour? Fetch my favorite.
Append methods: Do me a favour? Fetch my favorite.
```

The `replaceAll()` method handles the case of making the same change all through a string. If you want to change each matching occurrence to a different value, you can use `replaceFirst()` in a loop, as in Example 4-7. Here we make a pass through an entire string, turning each occurrence of either `cat` or `dog` into `feline` or `canine`. This is simplified from a real example that looked for *bit.ly* URLs and replaced them with the actual URL; the `computeReplacement` method there used the network client code from Recipe 14.1.

Example 4-7. main/src/main/java/regex/ReplaceMulti.java

```
/**
 * To perform multiple distinct substitutions in the same String,
 * you need a loop, and must call reset() on the matcher.
 */
public class ReplaceMulti {
  public static void main(String[] args) {

    Pattern patt = Pattern.compile("cat|dog");
    String line = "The cat and the dog never got along well.";
    System.out.println("Input: " + line);
    Matcher matcher = patt.matcher(line);
    while (matcher.find()) {
      String found = matcher.group(0);
      String replacement = computeReplacement(found);
      line = matcher.replaceFirst(replacement);
      matcher.reset(line);
    }
    System.out.println("Final: " + line);
  }

  static String computeReplacement(String in) {
    switch(in) {
    case "cat": return "feline";
    case "dog": return "canine";
    default: return "animal";
    }
  }
}
```

If you need to refer to portions of the occurrence that matched the regex, you can mark them with extra parentheses in the pattern and refer to the matching portion with `$1`, `$2`, and so on in the replacement string. Example 4-8 uses this to interchange two fields, in this case, turn names in the form `Firstname Lastname` into `Lastname, FirstName`.

Example 4-8. main/src/main/java/regex/ReplaceDemo2.java

```
public class ReplaceDemo2 {
  public static void main(String[] argv) {

    // Make an RE pattern
    String patt = "(\\w+)\\s+(\\w+)";

    // A test input.
    String input = "Ian Darwin";
    System.out.println("Input: " + input);

    // Run it from an RE instance and see that it works
    Pattern r = Pattern.compile(patt);
    Matcher m = r.matcher(input);
    m.find();
    System.out.println("Replaced: " + m.replaceFirst("$2, $1"));

    // The short inline version:
    // System.out.println(input.replaceFirst("(\\w+)\\s+(\\w+)", "$2, $1"));
  }
}
```

4.6 Printing All Occurrences of a Pattern

Problem

You need to find all the strings that match a given regex in one or more files or other sources.

Solution

Compare each line against the regex pattern.

Discussion

This example reads through a file one line at a time. Whenever a match is found, I extract it from the `line` and print it.

This code takes the `group()` methods from Recipe 4.4, the `substring` method from the `CharacterIterator` interface, and the `match()` method from the regex and

simply puts them all together. I coded it to extract all the names from a given file; in running the program through itself, it prints the words `import`, `java`, `until`, `regex`, and so on, each on its own line:

```
C:\> java ReaderIter.java ReaderIter.java
import
java
util
regex
import
java
io
Print
all
the
strings
that
match
given
pattern
from
file
public
...
C:\\>
```

I interrupted it here to save paper. This can be written two ways: a line-at-a-time pattern shown in Example 4-9 and a more efficient form using new I/O shown in Example 4-10 (the new I/O package used in both examples is described in Chapter 10).

Example 4-9. main/src/main/java/regex/ReaderIter.java

```
public class ReaderIter {
  public static void main(String[] args) throws IOException {
    // The RE pattern
    Pattern patt = Pattern.compile("[A-Za-z][a-z]+");
    // See the I/O chapter
    // For each line of input, try matching in it.
    Files.lines(Path.of(args[0])).forEach(line -> {
      // For each match in the line, extract and print it.
      Matcher m = patt.matcher(line);
      while (m.find()) {
        // Simplest method:
        // System.out.println(m.group(0));

        // Get the starting position of the text
        int start = m.start(0);
        // Get ending position
        int end = m.end(0);
        // Print whatever matched.
```

```
        // Use CharacterIterator.substring(offset, end);
        System.out.println(line.substring(start, end));
      }
    });
  }
}
```

Example 4-10. main/src/main/java/regex/GrepNIO.java

```
public class GrepNIO {
  public static void main(String[] args) throws IOException {

    if (args.length < 2) {
      System.err.println("Usage: GrepNIO patt file [...]");
      System.exit(1);
    }

    Pattern p=Pattern.compile(args[0]);
    for (int i=1; i<args.length; i++)
      process(p, args[i]);
  }

  static void process(Pattern pattern, String fileName) throws IOException {

    // Get a FileChannel from the given file.
    FileInputStream fis = new FileInputStream(fileName);
    FileChannel fc = fis.getChannel();

    // Map the file's content
    ByteBuffer buf = fc.map(FileChannel.MapMode.READ_ONLY, 0, fc.size());

    // Decode ByteBuffer into CharBuffer
    CharBuffer cbuf =
      Charset.forName("ISO-8859-1").newDecoder().decode(buf);

    Matcher m = pattern.matcher(cbuf);
    while (m.find()) {
      System.out.println(m.group(0));
    }
    fis.close();
  }
}
```

The nonblocking I/O version shown in Example 4-10 uses the new I/O (NIO) package java.nio, and relies on the fact that a nonblocking I/O `Buffer` can be used as a `CharSequence`. This program is more general in that the pattern argument is taken from the command-line argument. It prints the same output as the previous example if invoked with the pattern argument from the previous program on the command line:

```
java regex.GrepNIO "[A-Za-z][a-z]+"  ReaderIter.java
```

You might think of using `\w+` as the pattern; the only difference is that my pattern looks for well-formed capitalized words, whereas `\w+` would include Java-centric oddities like `theVariableName`, which have capitals in nonstandard positions.

Also note that the NIO version will probably be more efficient because it doesn't reset the `Matcher` to a new input source on each line of input as `ReaderIter` does.

4.7 Controlling Case in Regular Expressions

Problem

You want to find text regardless of case.

Solution

Use the `CASE_INSENSITIVE` option of the `Pattern.compile()` method.

Discussion

Compile the `Pattern` passing in the `flags` argument `Pattern.CASE_INSENSITIVE` to indicate that matching should be case-independent (i.e., that it should fold, ignore differences in case). If your code might run in different locales (see Recipe 3.12), then you should add `Pattern.UNICODE_CASE`. Without these flags, the default is normal, case-sensitive matching behavior. This flag (and others) is passed to the `Pattern.compile()` method, like this:

```
// regex/CaseMatch.java
Pattern  reCaseInsens = Pattern.compile(pattern, Pattern.CASE_INSENSITIVE |
    Pattern.UNICODE_CASE);
reCaseInsens.matches(input);        // will match case-insensitively
```

This flag must be passed when you create the `Pattern`; because `Pattern` objects are immutable, they cannot be changed once constructed.

The full source code for this example is online as *CaseMatch.java*.

Pattern.compile() Flags

Half a dozen flags can be passed as the second argument to `Pattern.compile()`. If more than one value is needed, they can be or'd together using the bitwise or operator `|`. In alphabetical order, these are the flags:

`CANON_EQ`
: Enables so-called canonical equivalence. In other words, characters are matched by their base character so that the character `e` followed by the combining character mark for the acute accent (´) can be matched either by the composite

character é or the letter e followed by the character mark for the accent (see Recipe 4.8).

`CASE_INSENSITIVE`
: Turns on case-insensitive matching (see Recipe 4.7).

`COMMENTS`
: Causes whitespace and comments (from # to end-of-line) to be ignored in the pattern. See *CommentedRegEx.java* in the *regex* source directory.

`DOTALL`
: Allows dot (.) to match any regular character or the newline, not just any regular character other than newline (see Recipe 4.9).

`MULTILINE`
: Specifies multiline mode (see Recipe 4.9).

`UNICODE_CASE`
: Enables Unicode-aware case folding (see Recipe 4.7).

`UNIX_LINES`
: Makes `\n` the only valid newline sequence for `MULTILINE` mode (see Recipe 4.9).

4.8 Matching Accented, or Composite, Characters

Problem

You want characters to match regardless of the form in which they are entered.

Solution

Compile the `Pattern` with the `flags` argument `Pattern.CANON_EQ` for canonical equality.

Discussion

Composite characters can be entered in various forms. Consider, as a single example, the letter e with an acute accent. This character can be found in various forms in Unicode text, such as the single character é (Unicode character `\u00e9`) or the two-character sequence e´ (e followed by the Unicode combining acute accent, `\u0301`). To allow you to match such characters regardless of which of possibly multiple fully decomposed forms are used to enter them, the regex package has an option for *canonical matching*, which treats any of the forms as equivalent. This option is enabled by passing `CANON_EQ` as (one of) the flags in the second argument to `Pattern.compile()`. Example 4-11 shows `CANON_EQ` being used to match several forms.

Example 4-11. main/src/main/java/regex/CanonEqDemo.java

```
public class CanonEqDemo {
  public static void main(String[] args) {
    String pattStr = "\u00e9gal"; // egal
    String[] input = {
        "\u00e9gal", // egal - this one had better match :-)
        "e\u0301gal", // e + "Combining acute accent"
        "e\u02cagal", // e + "modifier letter acute accent"
        "e'gal", // e + single quote
        "e\u00b4gal", // e + Latin-1 "acute"
    };
    Pattern pattern = Pattern.compile(pattStr, Pattern.CANON_EQ);
    for (int i = 0; i < input.length; i++) {
      if (pattern.matcher(input[i]).matches()) {
        System.out.println(pattStr + " matches input " + input[i]);
      } else {
        System.out.println(pattStr + " doesn't match input " + input[i]);
      }
    }
  }
}
```

This program correctly matches the combining accent and rejects the other characters, some of which, unfortunately, look like the accent on a printer, but are not considered combining accent characters:

```
égal matches input égal
égal matches input e?gal
égal does not match input e?gal
égal does not match input e'gal
égal does not match input e´gal
```

For more details, see the Unicode character charts.

4.9 Matching Newlines in Text

Problem

You need to match newlines in text.

Solution

Use `\n` or `\r` in your regex pattern. See also the flags constant `Pattern.MULTILINE`, which makes newlines match as beginning-of-line and end-of-line (`^` and `$`).

Discussion

Though line-oriented tools from Unix such as `sed` and `grep` match regular expressions one line at a time, not all tools do. The `Sam` text editor from Bell Laboratories was the first interactive tool I know of to allow multiline regular expressions; the Perl scripting language followed shortly after. In the Java API, the newline character by default has no special significance. The `BufferedReader` method `readLine()` normally strips out whichever newline characters it finds. If you read in gobs of characters using some method other than `readLine()`, you may have some number of `\n`, `\r`, or `\r\n` sequences in your text string.[4] Normally all of these are treated as equivalent to `\n`. If you want only `\n` to match, use the `UNIX_LINES` flag to the `Pattern.compile()` method.

In Unix, `^` and `$` are commonly used to match the beginning or end of a line, respectively. In this API, the regex metacharacters `^` and `$` ignore line terminators and only match at the beginning and the end, respectively, of the entire string. However, if you pass the `MULTILINE` flag into `Pattern.compile()`, these expressions match just after or just before, respectively, a line terminator; `$` also matches the very end of the string. Because the line ending is just an ordinary character, you can match it with `.` or similar expressions; and, if you want to know exactly where it is, `\n` or `\r` in the pattern match it as well. In other words, to this API, a newline character is just another character with no special significance (see the sidebar "Pattern.compile() Flags" on page 171). An example of newline matching is shown in Example 4-12.

Example 4-12. main/src/main/java/regex/NLMatch.java

```
public class NLMatch {
  public static void main(String[] argv) {

    String input = "I dream of engines\nmore engines, all day long";
    System.out.println("INPUT: " + input);
    System.out.println();

    String[] patt = {
      "engines.more engines",
      "ines\nmore",
      "engines$"
    };

    for (int i = 0; i < patt.length; i++) {
      System.out.println("PATTERN " + patt[i]);
```

4 Or a few related Unicode characters, including the next-line (`\u0085`), line-separator (`\u2028`), and paragraph-separator (`\u2029`) characters.

```
        boolean found;
        Pattern p1l = Pattern.compile(patt[i]);
        found = p1l.matcher(input).find();
        System.out.println("DEFAULT match " + found);

        Pattern pml = Pattern.compile(patt[i],
          Pattern.DOTALL|Pattern.MULTILINE);
        found = pml.matcher(input).find();
        System.out.println("MultiLine match " + found);
        System.out.println();
    }
  }
}
```

If you run this code, the first pattern (with the wildcard character .) always matches, whereas the second pattern (with $) matches only when `MATCH_MULTILINE` is set:

```
> java regex.NLMatch
INPUT: I dream of engines
more engines, all day long

PATTERN engines
more engines
DEFAULT match true
MULTILINE match: true

PATTERN engines$
DEFAULT match false
MULTILINE match: true
```

4.10 Program: Full Grep

Now that we've seen how the regular expressions package works, it's time to write JGrep, a full-blown version of the line-matching program with option parsing. Table 4-2 lists some typical command-line options that a Unix implementation of `grep` might include. For those not familiar with `grep`, it is a command-line tool that searches for regular expressions in text files. There are three or four programs in the standard `grep` family, and several newer replacements such as `ripgrep`, or `rg`. This program is my addition to this family of programs.

Table 4-2. `Grep` command-line options

Option	Meaning
`-c`	Count only; don't print lines, just count them
`-C`	Context; print some lines above and below each line that matches (not implemented in this version; left as an exercise for the reader)
`-f` *`pattern`*	Take pattern from file named after `-f` instead of from command line
`-h`	Suppress printing filename ahead of lines

Option	Meaning
`-i`	Ignore case
`-l`	List filenames only: don't print lines, just the names they're found in
`-n`	Print line numbers before matching lines
`-r`	Recursive mode (also allowed at `-R`)
`-s`	Suppress printing certain error messages
`-v`	Invert: print only lines that do NOT match the pattern

The Unix world features several getopt library routines for parsing command-line arguments, so I have a reimplementation of this in Java. As usual, because `main()` runs in a static context but our application main line does not, we could wind up passing a lot of information into the constructor. To save space, this version just uses global variables to track the settings from the command line. Unlike the Unix `grep` tool, this one does not yet handle combined options, so `-l -r -i` is OK, but `-lri` will fail, due to a limitation in the `GetOpt` parser used.

The program basically just reads lines, matches the pattern in them, and, if a match is found (or not found, with `-v`), prints the line (and optionally some other stuff, too). To save space, the code is not shown here, as it largely combines techniques shown previously. It remains available in *darwinsys-api/src/main/java/com/darwinsys/regex/JGrep.java* or online at my GitHub repository.

CHAPTER 5

Numbers

5.0 Introduction

Numbers are basic to just about any computation. They're used for array indices, temperatures, salaries, ratings, and an infinite variety of things. Yet they're not as simple as they seem. With floating-point numbers, how accurate is accurate? With random numbers, how random is random? With strings that should contain a number, what actually constitutes a number?

Java has eight built-in, or *primitive*, types, seven of which can be used to represent numbers. These are summarized in Table 5-1 with their *wrapper* (object) types, as well as two numeric classes that do not represent primitive types. The eighth primitive, `boolean`, and its wrapper `Boolean`, aren't numeric and can't be treated as such. Note that unlike languages such as C or Perl, which don't specify the size or precision of numeric types, Java—with its goal of portability—specifies these exactly and states that they are the same on all platforms.

Table 5-1. Numeric types

Built-in type	Object wrapper	Size of built-in (bits)	Contents
`byte`	`Byte`	8	Signed integer
`short`	`Short`	16	Signed integer
`int`	`Integer`	32	Signed integer
`long`	`Long`	64	Signed integer
`float`	`Float`	32	IEEE-754 floating point (single precision)
`double`	`Double`	64	IEEE-754 floating point (double precision)
`char`	`Character`	16	Unsigned Unicode character
n/a	`BigInteger`	unlimited	Arbitrary-size immutable integer value

Built-in type	Object wrapper	Size of built-in (bits)	Contents
n/a	`BigDecimal`	unlimited	Arbitrary-size-and-precision immutable floating-point value

As you can see, Java provides a numeric type for just about any purpose. There are four sizes of signed integers for representing various sizes of whole numbers. There are two sizes of floating-point numbers to approximate real numbers. There is also a type specifically designed to represent and allow operations on Unicode characters. The primitive numeric types are discussed here. The `Big` value types are described in Recipe 5.13.

When you read a string representing a number from user input or a text file, you need to convert it to the appropriate type. The object wrapper classes in the second column have several functions, one of which is to provide this basic conversion functionality—replacing the C programmer's `atoi`/`atof` family of functions and the numeric arguments to `scanf`.

Going the other way, you can convert any number to a string just by using string concatenation, or by using the `static` method `toString(value)` in each of the wrapper classes. If you want full control, Recipe 5.5 shows the use of `NumberFormat` and its related classes to provide full control of formatting.

As the name *object wrapper* implies, these classes are also used to wrap a number in a Java object, as many parts of the standard API are defined in terms of objects. Later on, "Solution" on page 504 shows using an `Integer` object to save an `int`'s value to a file using object serialization and retrieving the value later.

But I haven't yet mentioned the issues of floating point. Real numbers, you may recall, are numbers with a fractional part. There is an infinite number of real numbers. A floating-point number—what a computer uses to approximate a real number—is not the same as a real number. The number of floating-point numbers is finite, with only 2^32 different bit patterns for `float`s, and 2^64 for `double`s. Thus, most real values have only an approximate correspondence to floating point. The result of printing the real number 0.3 works correctly, like this:

```
// numbers/RealValues.java
System.out.println("The real value 0.3 is " + 0.3);
```

That code results in this printout:

```
The real value 0.3 is 0.3
```

But the difference between a real value and its floating-point approximation can accumulate if the value is used in a computation; this is often called a *rounding error*. Continuing the previous example, the real 0.3 multiplied by 3 yields:

```
The real 0.3 times 3 is 0.89999999999999991
```

Surprised? Not only is it off by a bit from what you might expect, but you will get the same output on any conforming Java implementation. I ran it on machines as disparate as an AMD/Intel PC with OpenBSD, a PC with Windows and the standard JDK, and on macOS, in Java versions from 8 to 22. Always the same answer. Whereas 3.0 * 3 gives 9.0, as you'd expect. And the same occurs in the C programming language, and should occur in any language that conforms to the IEEE Standard 754 for floating-point computation.

And what about random numbers? How random are they? You have probably heard the term *pseudorandom number generator*, or PRNG. All conventional random number generators, whether written in Fortran, C, or Java, generate pseudorandom numbers. That is, they're not truly random! True randomness comes only from specially built hardware: an analog source of Brownian noise connected to an analog-to-digital converter, for example.[1] Your average PC of today may have some good sources of entropy, or even hardware-based sources of randomness (which have not been widely used or tested yet). However, pseudorandom number generators are good enough for most purposes (other than cryptography, where special care is needed). Java provides one random generator in the base library java.lang.Math, and several others; we'll examine these in Recipe 5.9.

The class `java.lang.Math` contains an entire math library in one class, including trigonometry, conversions (including degrees to radians and back), rounding, truncating, square root, minimum, and maximum. It's all there. Check the Javadoc for details on the math library.

The package java.math contains support for *big numbers*—those larger than the normal built-in long integers, for example. See Recipe 5.13.

Java works hard to ensure that your programs are reliable. The usual ways you'd notice this are in the common requirement to catch potential exceptions—all through the Java API—and in the need to *cast*, or convert, when storing a value that might or might not fit into the variable you're trying to store it in. I'll show examples of these.

Overall, Java's handling of numeric data fits well with the ideals of portability, reliability, and ease of programming.

1 For a low-cost source of randomness, check out the now-defunct Lavarand. The process used digitized video of 1970s lava lamps to provide "hardware-based" randomness. Fun!

See Also

The Java Language Specification, and the Javadoc page for `java.lang.Math`.

5.1 Checking Whether a String Is a Valid Number

Problem

You need to check whether a given string contains a valid number, and, if so, convert it to binary (internal) form.

Solution

To accomplish this, either use the appropriate wrapper class's conversion routine and catch the `NumberFormatException`, or try to match with a regular expression (Chapter 4). The code in Example 5-1 converts a string to a `double`.

Example 5-1. main/src/main/java/numbers/StringToDouble.java

```
public static void main(String[] argv) {
  String aNumber = argv[0];  // not argv[1]
  double result;
  try {
    result = Double.parseDouble(aNumber);
    System.out.println("Number is " + result);
  } catch(NumberFormatException exc) {
    System.out.println("Invalid number " + aNumber);
    return;
  }
}
```

Discussion

This code (with suitable change for any other numeric wrapper class) lets you validate numbers in the format that the designers of the wrapper classes expected. If you need to accept a different definition of numbers, you could use regular expressions (see Chapter 4) to make the determination.

There may also be times when you want to tell whether a given numeric string contains an integer number or a floating-point number. One way is to check for the characters `.`, `d`, `e`, or `f` in the input; if one of these characters is present, convert the number as a `double`. Otherwise, convert it as an `int`, as in Example 5-2. Another possibility is to convert it to a `Double` (using `valueOf(String)`), then test `myDouble.intValue()==myDouble.doubleValue()`, which should always be true if there is no decimal fraction.

Example 5-2. main/src/main/java/numbers/GetNumber.java

```
/*
 * Process one String, displaying it as int or double as appropriate
 */
public static Number process(String s) {
  if (s.matches("[+-]*\\d*\\.\\d+[dDeEfF]*")) {
    try {
      double dValue = Double.parseDouble(s);
      System.out.println(s + ": is a double: " + dValue +
        " by Double.parseDouble()");
      return dValue;
    } catch (NumberFormatException e) {
      System.out.println(s + ": Invalid double: " + s);
      return Double.NaN;
    }
  } else try {
    // did not contain . d e or f, so try as long.
      long longValue = Long.parseLong(s);
      System.out.println(s + ": is a long: " + longValue +
        " by Long.parseLong()");
      return longValue;
    } catch (NumberFormatException e2) {
      System.out.println(s + ": Not a number: " + s);
      return Double.NaN;
    }
}

public boolean isIntegerNumber(String s) {
  Double d = Double.valueOf(s);
  boolean result = d.intValue() == d.doubleValue();
  System.out.println(s + ": is " + (result ?
    "an integer" : "a floating-point number") +
    " by Double.xxxValue() comparison");
  return result;
}
```

See Also

A more involved form of parsing is offered by the `DecimalFormat` class, discussed in Recipe 5.5. There is also the `Scanner` class; see Recipe 10.7.

5.2 Converting Numbers to Objects and Vice Versa

Problem

You need to convert numbers to objects and objects to numbers.

Solution

Use the object wrapper classes listed in Table 5-1 at the beginning of this chapter.

Discussion

Often you have a primitive number and you need to pass it into a method where an `Object` is required, or vice versa. Long ago you had to invoke the conversion routines that are part of the `wrapper` classes, but now you can generally use automatic conversion (called *auto-boxing/auto-unboxing*). See Example 5-3 for examples of both.

Example 5-3. main/src/main/java/structure/AutoboxDemo.java

```
public class AutoboxDemo {

    /** Shows auto-boxing (in the call to foo(i), i is wrapped automatically)
     * and auto-unboxing (the return value is automatically unwrapped).
     */
    public static void main(String[] args) {
        int i = 42;
        int result = foo(i);            ❶
        System.out.println(result);
    }

    public static Integer foo(Integer i) {
        System.out.println("Object = " + i);
        return Integer.valueOf(123);    ❷
    }
}
```

❶ Auto-boxing: `int 42` is converted to `Integer(42)`. Also auto-unboxing: the `Integer` returned from `foo()` is auto-unboxed to assign to `int result`.

❷ No auto-boxing: `valueOf()` returns `Integer`. If the line said `return Integer.intValueOf(123)`, then it would be a second example of auto-boxing because the method return value is `Integer`.

To explicitly convert between an `int` and an `Integer` object, or vice versa, you can use the `wrapper` class methods:

```
// int to Integer
Integer wrapped = Integer.valueOf(42);
System.out.println(wrapped.toString());    // or just "wrapped"

// Integer to int
int primitive = wrapped.intValue();
System.out.println(primitive);
```

Any boxed value of `boolean`, `byte`, `char` up to 127, and `short` or `int` from −128 to +127, will be *interned*, that is, only a single instance will exist no matter how many times you do the boxing conversion of a given value. This may give unexpected results when using == comparisons.

5.3 Taking a Fraction of an Integer Without Using Floating Point

Problem

You want to multiply an integer by a fraction without converting the fraction to a floating-point number.

Solution

Multiply the integer by the numerator and divide by the denominator.

This technique should be used only when efficiency is more important than clarity because it tends to detract from the readability—and therefore the maintainability—of your code.

Discussion

Because integers and floating-point numbers are stored differently, it may sometimes be desirable and feasible, for efficiency purposes, to multiply an integer by a fractional value without converting the values to floating point and back, and without requiring a cast. This is shown in Example 5-4.

Example 5-4. main/src/main/java/numbers/FractMult.java

```
public class FractMult {
    public static void main(String[] u) {

        double d1 = 0.666 * 5;  // fast but obscure and inaccurate: convert
        System.out.println(d1); // 2/3 to 0.666 in programmer's head

        double d2 = 2/3 * 5;    // wrong answer - 2/3 == 0, 0*5 = 0
```

```
        System.out.println(d2);

        double d3 = 2d/3d * 5;  // "normal"
        System.out.println(d3);

        double d4 = (2*5)/3d;   // one step done as integers, almost same answer
        System.out.println(d4);

        int i5 = 2*5/3;         // fast, approximate integer answer
        System.out.println(i5);
    }
}
```

Running the code looks like this:

```
$ java numbers.FractMult
3.33
0.0
3.333333333333333
3.3333333333333335
3
$
```

You should beware of the possibility of numeric overflow and avoid this optimization if you cannot guarantee that the multiplication by the numerator will not overflow.

5.4 Working with Floating-Point Numbers

Problem

You want to be able to compare and round floating-point numbers.

Solution

Compare with the `INFINITY` constants, and use `isNaN()` to check for `NaN` (not a number).

Compare floating values with an epsilon value.

Round floating-point values with `Math.round()` or custom code.

Discussion

Comparisons can be a bit tricky. Fixed-point (`short`, `int`, `long`) operations that divide by zero result in Java notifying you abruptly by throwing an exception. This is because integer division by zero is considered a *logic error*.

Floating-point operations, however, do not throw an exception because they are defined over an (almost) infinite range of values. Instead, they signal errors by

producing the constant `POSITIVE_INFINITY` if you divide a positive floating-point number by zero, the constant `NEGATIVE_INFINITY` if you divide a negative floating-point value by zero, and `NaN` if you otherwise generate an invalid result. Values for these three public constants are defined in both the `Float` and the `Double` wrapper classes. The value `NaN` has the unusual property that it is not equal to itself (i.e., `NaN != NaN`). Thus, it would hardly make sense to compare a (possibly suspect) number against `NaN`, because the following expression can never be true:

```
x == NaN
```

Instead, the methods `Float.isNaN(float)` and `Double.isNaN(double)` must be used, as in Example 5-5.

Example 5-5. main/src/main/java/numbers/InfNaN.java

```
public static void main(String[] argv) {
  double d = 123;
  double e = 0;
  if (d/e == Double.POSITIVE_INFINITY)
    System.out.println("Check for POSITIVE_INFINITY works");
  double s = Math.sqrt(-1);
  if (s == Double.NaN)
    System.out.println("Comparison with NaN incorrectly returns true");
  if (Double.isNaN(s))
    System.out.println("Double.isNaN() correctly returns true");
}
```

Note that this, by itself, is not sufficient to ensure that floating-point calculations have been done with adequate accuracy. For example, the following program demonstrates a contrived calculation—Heron's formula for the area of a triangle—both in `float` and in `double`. The double values are correct, but the floating-point value comes out as zero due to rounding errors. This happens because, in Java, operations involving only `float` values are performed as 32-bit calculations. Related languages such as C automatically promote these to double during the computation, which can eliminate some loss of accuracy. Let's take a look:

```
public class Heron {
  public static void main(String[] args) {
    // Sides for triangle in float
    float af, bf, cf;
    float sf, areaf;

    // Ditto in double
    double ad, bd, cd;
    double sd, aread;

    // Area of triangle in float
    af = 12345679.0f;
```

```
        bf = 12345678.0f;
        cf = 1.01233995f;

        sf = (af+bf+cf)/2.0f;
        areaf = (float)Math.sqrt(sf * (sf - af) * (sf - bf) * (sf - cf));
        System.out.println("Single precision: " + areaf);

        // Area of triangle in double
        ad = 12345679.0;
        bd = 12345678.0;
        cd = 1.01233995;

        sd = (ad+bd+cd)/2.0d;
        aread = Math.sqrt(sd * (sd - ad) * (sd - bd) * (sd - cd));
        System.out.println("Double precision: " + aread);
    }
}
```

Now let's run it:

```
$ java numbers.Heron
Single precision: 0.0
Double precision: 972730.0557076167
$
```

If in doubt, use `double`!

Prior to Java 17, there were potential issues around large-magnitude `double` computations that might give different results on different Java implementations. Java provided the keyword `strictfp`, which can apply to classes, interfaces, or methods within a class.[2] If a computation is Strict-FP, then it must always, for example, return the value `INFINITY` if a calculation would overflow the value of `Double.MAX_VALUE` (or underflow the value `Double.MIN_VALUE`). Non-Strict-FP calculations—formerly the default—were allowed to perform calculations on a greater range and can return a valid final result that is in range even if the interim product is out of range. This is pretty esoteric and affects only computations that approach the bounds of what fits into a double. Java 17 changed the default for all floating-point computations to Strict-FP; thus the `strictfp` keyword, while still allowed, no longer has any effect.

2 Note that an expression consisting entirely of compile-time constants, like `Math.PI * 2.1e17`, was also considered to be Strict-FP.

Comparing floating-point values

Based on what we've just discussed, you probably won't just go comparing two floats or doubles for equality. You might expect the floating-point wrapper classes, `Float` and `Double`, to override the `equals()` method, which they do. The `equals()` method returns `true` if the two values are the same bit for bit (i.e., if and only if the numbers are the same or are both `NaN`). It returns `false` otherwise, including if the argument passed in is null, or if one object is +0.0 and the other is –0.0.

I said earlier that `NaN != NaN`, but if you compare with `equals()`, the result is true:

```
jshell> Float f1 = Float.valueOf(Float.NaN)
f1 ==> NaN

jshell> Float f2 = Float.valueOf(Float.NaN)
f2 ==> NaN

jshell> f1 == f2 # Comparing object identities
$4 ==> false

jshell> f1.equals(f1) # bitwise comparison of values
$5 ==> true
```

If this sounds weird, remember that the complexity comes partly from the nature of doing real number computations in the less-precise floating-point hardware. It also comes partly from the details of the IEEE Standard 754, which specifies the floating-point functionality that Java tries to adhere to so that underlying floating-point processor hardware can be used even when Java programs are being interpreted.

To actually compare floating-point numbers for equality, it is generally desirable to compare them within some tiny range of allowable differences; this range is often regarded as a tolerance or as *epsilon*. Example 5-6 shows an `equals()` method you can use to do this comparison, as well as comparisons on values of `NaN`. When run, it prints that the first two numbers are equal within epsilon:

```
$ java numbers.FloatCmp
True within epsilon 1.0E-7
$
```

Example 5-6. main/src/main/java/numbers/FloatCmp.java

```
public class FloatCmp {

  final static double EPSILON = 0.0000001;

  public static void main(String[] argv) {
    double da = 3 * .3333333333;
    double db = 0.99999992857;

    // Compare two numbers that are expected to be close.
```

```
    if (da == db) {
      System.out.println("Java considers " + da + "==" + db);
    // else compare with our own equals overload
    } else if (equals(da, db, 0.0000001)) {
      System.out.println("Equal within epsilon " + EPSILON);
    } else {
      System.err.println(da + " != " + db);
    }

    System.out.println("NaN prints as " + Double.NaN);

    // Show that comparing two NaNs is not a good idea:
    double nan1 = Double.NaN;
    double nan2 = Double.NaN;
    if (nan1 == nan2)
      System.out.println("Comparing two NaNs incorrectly returns true.");
    else
      System.err.println("Comparing two NaNs correctly reports false.");

    if (Double.valueOf(nan1).equals(Double.valueOf(nan2)))
      System.out.println("Double(NaN).equals(NaN) correctly returns true.");
    else
      System.err.println("Double(NaN).equals(NaN) incorrectly returns false.");
  }

  /** Compare two doubles within a given epsilon */
  public static boolean equals(double a, double b, double eps) {
    if (a==b) return true;
    // If the difference is less than epsilon, treat as equal.
    return Math.abs(a - b) < eps;
  }

  /** Compare two doubles, using default epsilon */
  public static boolean equals(double a, double b) {
    return equals(a, b, EPSILON);
  }
}
```

Note that none of the `System.err` messages about incorrect returns prints. The point of this example with `NaN`s is that you should always make sure values are not `NaN` before entrusting them to `Double.equals()`.

Rounding

If you simply cast a floating value to an integer value, Java truncates the value. A value like 3.999999 cast to an `int` or `long` becomes 3, not 4. To round floating-point numbers properly, use `Math.round()`. It has two overloads: if you give it a `double`, you get a `long` result; if you give it a `float`, you get an `int`.

What if you don't like the rounding rules used by `round`? If, for some strange reason, you wanted to round numbers greater than 0.54 instead of the normal 0.5, you could write your own version of `round()`:

```
public class Round {
  /** We round a number up if its fraction exceeds this threshold. */
  public static final double THRESHOLD = 0.54;

  /*
   * Round floating values to integers.
   * @return the closest int to the argument.
   * @param d A non-negative value to be rounded.
   */
  public static int round(double d) {
    return (int)Math.floor(d + 1.0 - THRESHOLD);
  }

  public static void main(String[] argv) {
    for (double d = 0.1; d<=1.0; d+=0.05) {
      System.out.println("My way:  " + d + "-> " + round(d));
      System.out.println("Math way:" + d + "-> " + Math.round(d));
    }
  }
}
```

If, on the other hand, you simply want to display a number with less precision than it normally gets, you probably want to use a `DecimalFormat` object, which we look at in Recipe 5.5, or use a `printf`-like format code, discussed in Recipe 3.2.

5.5 Formatting Numbers

Problem

You need to format numbers.

Solution

Use a `NumberFormat` subclass. Or use a `Formatter`.

Discussion

The C language `printf()` function provides many formatting abilities. Java did not originally provide C-style `printf/scanf` functions because they tend to mix together formatting and input/output in a very inflexible way. Programs using `printf/scanf` can be hard to internationalize, for example. Of course, by popular demand, Java did eventually introduce `printf()`, which along with `String.format()` is now standard in Java, based on an underlying `Formatter` class. See Recipe 3.2.

Java has an entire package, java.text, full of formatting routines as general and flexible as anything you might imagine. As with `printf`, it has an involved formatting language, described in the Javadoc page. Consider the presentation of long numbers. In North America, the number "one thousand twenty-four and a quarter" is written 1,024.25. In other parts of the world it is written 1 024,25 or 1.024,25. Not to mention how currencies and percentages are formatted! Trying to keep track of this yourself would drive the average small software shop around the bend rather quickly.

Fortunately, the java.util package includes a `Locale` class; and, furthermore, the Java runtime automatically sets a default `Locale` object based on the user's environment (on Macintosh and Windows, the user's preferences, and on Unix, the user's environment variables).

To provide a nondefault locale in code, see Recipe 3.12. To provide formatters customized for numbers, currencies, and percentages, the `NumberFormat` class has static *factory methods* that normally return a `DecimalFormat` with the correct pattern already instantiated. A `DecimalFormat` object appropriate to the user's locale can be obtained from the factory method `NumberFormat.getInstance()` and manipulated using `set` methods. Surprisingly, the method `setMinimumIntegerDigits()` turns out to be the easy way to generate a number format with leading zeros. Example 5-7 demonstrates this.

Example 5-7. main/src/main/java/numbers/NumFormat2.java

```
public class NumFormat2 {

  public static final double data[] = {
    0, 1, 22d/7, 100.2345678
  };

  public static void main(String[] av) {
    // Get a format instance
    NumberFormat form = NumberFormat.getInstance();

    // Set it to look like 999.99[99]
    form.setMinimumIntegerDigits(3);
    form.setMinimumFractionDigits(2);
    form.setMaximumFractionDigits(4);

    // Now print using it.
    for (double d : data) {
      System.out.println(
        d + "\tformats as " + form.format(d));
    }
  }
}
```

This prints the contents of the array using the `NumberFormat` instance `form`:

```
$ java numbers.NumFormat2
0.0     formats as 000.00
1.0     formats as 001.00
3.142857142857143       formats as 003.1429
100.2345678     formats as 100.2346
$
```

You can also construct a `DecimalFormat` with a particular pattern or change the pattern dynamically using `applyPattern()`. Some of the more common pattern characters are shown in Table 5-2.

Table 5-2. `DecimalFormat` pattern characters

Character	Meaning
`#`	Numeric digit (leading zeros suppressed)
`0`	Numeric digit (leading zeros provided)
`.`	Locale-specific decimal separator (decimal point)
`,`	Locale-specific grouping separator (comma in English)
`-`	Locale-specific negative indicator (minus sign)
`%`	Shows the value as a percentage
`;`	Separates two formats: the first for positive and the second for negative values
`'`	Escapes one of the preceding characters so it appears
Anything else	Appears as itself

The `NumFormatDemo` program uses one `DecimalFormat` to print a number with only two decimal places and a second to format the number according to the default locale, as shown in Example 5-8.

Example 5-8. main/src/main/java/numbers/NumFormatDemo.java

```
/** A number to format */
public static final double intlNumber = 1024.25;
/** Another number to format */
public static final double ourNumber = 100.2345678;
  NumberFormat defForm = NumberFormat.getInstance();
  NumberFormat ourForm = new DecimalFormat("##0.##");
  // toPattern() will reveal the combination of #0., etc
  // that this particular Locale uses to format with!
  System.out.println("defForm's pattern is " +
    ((DecimalFormat)defForm).toPattern());
  System.out.println(intlNumber + " formats as " +
    defForm.format(intlNumber));
  System.out.println(ourNumber + " formats as " +
    ourForm.format(ourNumber));
```

```
    System.out.println(ourNumber + " formats as " +
      defForm.format(ourNumber) + " using the default format");
```

This program prints the given pattern and then formats the same number using several formats:

```
$ java numbers.NumFormatDemo
defForm's pattern is
,
0.
1024.25 formats as 1,024.25
100.2345678 formats as 100.23
100.2345678 formats as 100.235 using the default format
$
```

Human-readable number formatting

12 To print a number in what Linux/Unix calls "human-readable format" (many display commands accept a `-h` argument for this format), use the Java 12 `CompactNumberFormat`, as shown in Example 5-9.

Example 5-9. nmain/src/main/java/numbers/CompactFormatDemo.java

```
public class CompactFormatDemo {

  static final Number[] nums = {
    0, 1, 1.25, 1234, 12345, 123456.78, 123456789012L
  };
  static final String[] strs = {
    "1", "1.25", "1234", "12.345K", "1234556.78", "123456789012L"
  };

  public static void main(String[] args) throws ParseException {
    NumberFormat cnf = NumberFormat.getCompactNumberInstance();
    System.out.println("Formatting:");
    for (Number n : nums) {
      cnf.setParseIntegerOnly(false);
      cnf.setMinimumFractionDigits(2);
      System.out.println(n + ": " + cnf.format(n));
    }
    System.out.println("Parsing:");
    for (String s : strs) {
      System.out.println(s + ": " + cnf.parse(s));
    }
  }

}
```

The output is shown here:

```
$ java CompactFormatDemo.java
Formatting:
0: 0
1: 1
1.25: 1
1234: 1.23K
12345: 12.35K
123456.78: 123.46K
123456789012: 123.46B
Parsing:
1: 1
1.25: 1.25
1234: 1234
12.345K: 12345
1234556.78: 1234556.78
123456789012L: 123456789012
$
```

Roman numeral formatting

This is presented as a good example of writing your own `Format` subclass for unusual formatting needs; it's obviously not that common to need Roman numerals these days. To work with Roman numerals, use my `RomanNumberFormat` class, as in Example 5-10.

Example 5-10. main/src/main/java/numbers/RomanNumberSimple.java

```
RomanNumberFormat rf = new RomanNumberFormat();
var year = LocalDate.now().getYear();
var yearStr = rf.format(year);
System.out.println(year + " -> " + yearStr);
long newYear = (Long) rf.parseObject(yearStr);
System.out.println(yearStr + " -> " + newYear);
if (newYear != year) {
  System.out.println("Error: non-idempotent!");
}
```

Running `RomanNumberSimple` in 2025 produces this output:

```
2025 -> MMXXV
MMXXV -> 2025
```

The source of the `RomanNumberFormat` class, not included here to save space, is in *darwinsys-api/src/main/java/com/darwinsys/numbers/RomanNumberFormat.java* and online at my GitHub repository. Some of the public methods are required because I wanted it to be a subclass of the abstract class `Format`, including three different `format()` overloads.

See Also

Java I/O by Elliotte Rusty Harold (O'Reilly) includes an entire chapter on `NumberFormat` and develops the subclass `ExponentialNumberFormat`.

5.6 Converting Among Binary, Octal, Decimal, and Hexadecimal

Problem

You want to display an integer as a series of bits—for example, when interacting with certain hardware devices—or in some alternative number base (binary is base 2, octal is base 8, decimal is 10, hexadecimal is 16). You want to convert a binary number or a hexadecimal value into an integer.

Solution

The class `java.lang.Integer` provides the solutions. Most of the time you can use `Integer.parseInt(String input)` or `Integer.parseInt(String input, int radix)` to convert from any string containing a number to an `Integer`, and `Integer.toString(int input)` or `Integer.toString(int input, int radix)` to convert an `int` value to a `String`. Example 5-11 shows some examples of using the `Integer` class.

Example 5-11. main/src/main/java/numbers/IntegerBinOctHexEtc.java

```
String input = "101010";
int i = 42;
System.out.println(input + " in default toString() is " + Integer.toString(i));
for (int radix : new int[] { 2, 8, 10, 16, 36 }) {
  System.out.print(input + " in base " + radix + " is "
      + Integer.parseInt(input, radix) + "; ");
  System.out.println(i + " formatted in base " + radix + " is "
      + Integer.toString(i, radix));
}
```

This program prints the binary string as an integer in various bases, and the integer 42 in those same number bases:

```
$ java numbers.IntegerBinOctHexEtc
101010 in default toString() is 42
101010 in base 2 is 42; 42 formatted in base 2 is 101010
101010 in base 8 is 33288; 42 formatted in base 8 is 52
101010 in base 10 is 101010; 42 formatted in base 10 is 42
101010 in base 16 is 1052688; 42 formatted in base 16 is 2a
```

```
101010 in base 36 is 60512868; 42 formatted in base 36 is 16
$
```

Discussion

There are also simpler versions of `toString(int)` that don't require you to specify the radix, for example, `toBinaryString()` to convert an integer to binary string representation, `toHexString()` to hexadecimal, `toOctalString()`, and so on. The Javadoc page for the `Integer` class is your friend here.

The `String` class itself includes a series of static methods—`valueOf(int)`, `valueOf(double)`, and so on—that also provide default formatting. That is, they return the given numeric value formatted as a string.

5.7 Operating on a Range of Integers

Problem

You need to work on a range of integers.

Solution

For a contiguous set, use `IntStream::range` and `rangeClosed`, or the older `for` loop.

For discontinuous ranges of numbers, use a `java.util.BitSet`. Or, just use a `List` (see Recipe 7.5).

Discussion

To process a contiguous set of integers, Java provides both `range()` / `rangeClosed()` methods in the `IntStream` and `LongStream` classes. These take a starting and ending number; `range()` excludes the ending number while `rangeClosed()` closes on, or includes, the ending number. You can also iterate over a range of numbers using the traditional `for` loop. Loop control for the `for` loop is in three parts: initialize, test, and change. If the test part is initially false, the loop will never be executed, not even once. You can iterate over the elements of an array or collection of numbers (or of any type) using a for-each loop (see Chapter 7).

The program in Example 5-12 demonstrates these techniques. The `->` syntax is a lambda expression; see Recipe 9.1 if you are not yet familiar with this syntax.

Example 5-12. main/src/main/java/numbers/NumSeries.java

```
public class NumSeries {
  public static void main(String[] args) {
```

```
    // For ordinal list of numbers n to m, use rangeClosed(start, endInclusive)
    IntStream.rangeClosed(1, 12).forEach(
      i -> System.out.println("Month # " + i));

    // Or, use a for loop starting at 1.
    for (int i = 1; i <= months.length; i++)
      System.out.println("Month # " + i);

    // Or a foreach loop
    for (String month : months) {
      System.out.println(month);
    }

    // When you want a set of array indices, use range(start, endExclusive)
    IntStream.range(0, months.length).forEach(
      i -> System.out.println("Month " + months[i]));

    // Or, use a for loop starting at 0.
    for (int i = 0; i < months.length; i++)
      System.out.println("Month " + months[i]);

    // For e.g., counting by 3 from 11 to 27, use a for loop
    for (int i = 11; i <= 27; i += 3) {
      System.out.println("i = " + i);
    }

    // A discontiguous set of integers, using a BitSet

    // Create a BitSet and turn on a couple of bits.
    BitSet b = new BitSet();
    b.set(0);  // January
    b.set(3);  // April
    b.set(8);  // September

    // Print the months
    for (int i = 0; i<months.length; i++) {
      if (b.get(i))
        System.out.println("Month " + months[i]);
    }
    // Shorter way, using IntStream stream() method
    b.stream().forEach(i->System.out.println(months[i]));

    // Same example but shorter:
    // a discontiguous set of integers, using an array
    int[] numbers = {0, 3, 8};

    // Presumably somewhere else in the code... Also a foreach loop
    for (int n : numbers) {
      System.out.println("Month: " + months[n]);
    }
  }
```

```
  /** Names of months. See Dates/Times chapter for a better way to get these */
  protected static String months[] = {
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
  };
}
```

5.8 Formatting with Correct Plurals

Problem

You're printing something like `"We used " + n + " items"`, but in English, "We used 1 items" is ungrammatical. You want "We used 1 item."

Solution

Use a `ChoiceFormat` or a conditional expression.

Discussion

Use Java's ternary operator (`cond ? trueval : falseval`) in a string concatenation. Both zero and plurals get an "s" appended to the noun in English ("no books, one book, two books"), so we test for `n==1`:

```
public class FormatPlurals {
  public static void main(String[] argv) {
    report(0);
    report(1);
    report(2);
  }

  /** report -- using conditional operator */
  public static void report(int n) {
    System.out.println("We used " + n + " item" + (n==1?"":"s"));
  }
}
```

Does it work?

```
$ java numbers.FormatPlurals
We used 0 items
We used 1 item
We used 2 items
$
```

The final `println` statement is effectively equivalent to the following:

```
if (n==1)
    System.out.println("We used " + n + " item");
```

```
    else
        System.out.println("We used " + n + " items");
```

This is a lot longer, so the ternary conditional operator is worth learning.

The `ChoiceFormat` is ideal for this. It is actually capable of much more, but here I'll show only this simplest use. I specify the values 0, 1, and 2 (or more), and the string values to print corresponding to each number. The numbers are then formatted according to the range they fall into, as shown in Example 5-13.

Example 5-13. main/src/main/java/numbers/FormatPluralsChoice.java

```
public class FormatPluralsChoice extends FormatPlurals {

  // ChoiceFormat to just give pluralized word
  static double[] limits = { 0, 1, 2 };
  static String[] formats = { "reviews", "review", "reviews"};
  static ChoiceFormat pluralizedFormat = new ChoiceFormat(limits, formats);

  // ChoiceFormat to give English text version, quantified
  static ChoiceFormat quantizedFormat = new ChoiceFormat(
    "0#no reviews|1#one review|1<many reviews");

  // Test data
  static int[] data = { -1, 0, 1, 2, 3 };

  public static void main(String[] argv) {
    System.out.println("Pluralized Format");
    for (int i : data) {
      System.out.println("Found " + i + " " + pluralizedFormat.format(i));
    }

    System.out.println("Quantized Format");
    for (int i : data) {
      System.out.println("Found " + quantizedFormat.format(i));
    }
  }
}
```

This generates the same output as the basic version. It is slightly longer, but more general, and lends itself better to internationalization.

See Also

In addition to `ChoiceFormat`, the same result can be achieved with a `MessageFormat`. The online source in file *main/src/main/java/i18n/MessageFormatDemo.java* has an example.

5.9 Generating Random Numbers

Problem

You need to generate pseudorandom numbers in a hurry.

Solution

Use `java.lang.Math.random()`.

Discussion

Use `java.lang.Math.random()` to generate random numbers. There is no promise that the random values it returns are very *good* random numbers, however. Like most software-only implementations, these are *pseudorandom number generators* (PRNGs), meaning that the numbers are not totally random, but devised from an algorithm. That said, they are adequate for casual use. This code exercises the `random()` method:

```
// java.lang.Math.random( ) is static, so you don't need any constructor calls
System.out.println("A random from java.lang.Math is " + Math.random( ));
```

Note that this method only generates double values. If you need integers, construct a `java.util.Random` object and call its `nextInt()` method; if you pass it an integer value, this will become the upper bound. Here I generate a bunch of integers from 1 to 10:

```
public class RandomInt {
  public static void main(String[] a) {
    Random r = new Random();
    for (int i=0; i<1000; i++)
      // nextInt(10) goes from 0-9; add 1 for 1-10;
      System.out.println(1+r.nextInt(10));
  }
}
```

To see if this `RandomInt` demo was really working well, I used the Unix tools `sort` and `uniq`, which together give a count of how many times each value was chosen. For 1,000 integers, each of 10 values should be chosen about 100 times. I ran it twice to get a better idea of the distribution:

```
$ java numbers.RandomInt | sort | uniq -c | sort -k 2 -n
  96 1
 107 2
 102 3
 122 4
  99 5
 105 6
  97 7
  96 8
```

```
  79 9
  97 10
$ java numbers.RandomInt | sort | uniq -c | sort -k 2 -n
  86 1
  88 2
 110 3
  97 4
  99 5
 109 6
  82 7
 116 8
  99 9
 114 10
$
```

Not bad, but the next step is to run these through a statistical program to see how random they really are; we'll return to this in a minute.

In general, to generate random numbers, you need to construct a `java.util.Random` object (not just any old random object) and call its `next*()` methods. These methods include `nextBoolean()`, `nextBytes()` (which fills the given array of bytes with random values), `nextDouble()`, `nextFloat()`, `nextInt()`, and `nextLong()`. Don't be confused by the capitalization of `Float`, `Double`, etc. They return the primitive types `boolean`, `float`, `double`, etc., not the capitalized wrapper objects. Clear enough? Maybe an example will help:

```
// java.util.Random methods are non-static; we need an instance
Random r = new Random();
for (int i=0; i<10; i++) {
  System.out.println("A double from java.util.Random is " + r.nextDouble());
}
for (int i=0; i<10; i++) {
  System.out.println("An integer from java.util.Random is " + r.nextInt());
}
```

A fixed value (*starting seed*) can be provided to generate repeatable values, as for testing. You can also use the `java.util.Random nextGaussian()` method, as shown next. The `nextDouble()` methods try to give a flat distribution between 0 and 1.0, in which each value has an equal chance of being selected. A Gaussian or normal distribution is a bell curve of values from negative infinity to positive infinity, with the majority of the values around zero (0.0):

```
// numbers/Random3.java
Random r = new Random();
for (int i = 0; i < 10; i++)
    System.out.println("A gaussian random double is " + r.nextGaussian());
```

To illustrate the different distributions, I generated 10,000 numbers first by using `nextRandom()` and then using `nextGaussian()`. The code for this is in *Random4.java* (not shown here) and is a combination of the previous programs with code to print

the results into files. I then plotted histograms using the R statistics package (see Chapter 12 and the R Project website). The R script used to generate the graph, *randomnesshistograms.r*, doesn't contain any Java code; it's in *javasrc* under *main/src/main/resources*. The results are shown in Figure 5-1.

The bar graph should be reasonably flat, and the Gaussian should resemble the hated-by-students "bell curve." Looks like both PRNGs do their job!

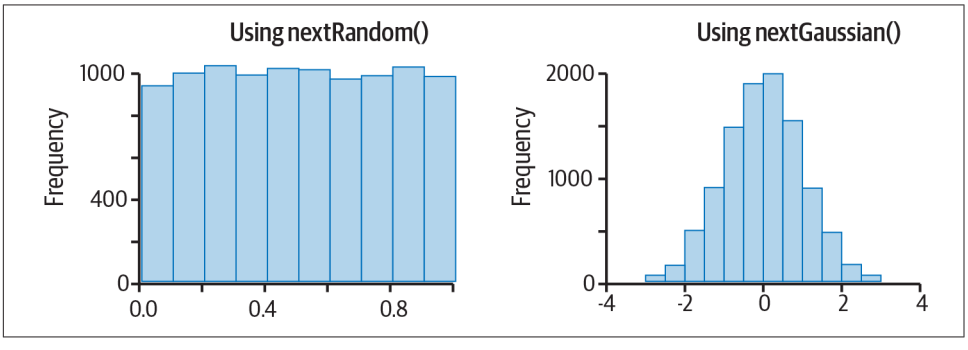


Figure 5-1. Flat (left) and Gaussian (right) distributions

See Also

For more information, see the Javadoc documentation for `java.util.Random`, and the discussion in the introdcution to Chapter 5 about pseudorandomness versus real randomness. This is also treated in my *Java Magazine* writeup on random numbers, where I give an example of "true" randomness via 1980s hardware.

For cryptographic use, see the class `java.security.SecureRandom`, which provides cryptographically strong pseudorandom number generators.

5.10 Multiplying Matrices

Problem

You need to multiply a pair of two-dimensional arrays, as is common in mathematical and engineering applications.

Solution

Use the following code with nested loops as a model.

Discussion

It is straightforward to multiply an array of a numeric type. In real life you would probably use a full-blown package such as the Efficient Java Matrix Library (EJML) or Deeplearning4j's ND4J package. However, a simple implementation can serve to show the concepts involved. The code in Example 5-14 implements matrix multiplication.

Example 5-14. main/src/main/java/numbers/Matrix.java

```
public class Matrix {

  /* Matrix-multiply two arrays together.
   * The arrays MUST be rectangular.
   * @author Adapted from Tom Christiansen & Nathan Torkington's
   * implementation in their Perl Cookbook.
   */
  public static int[][] multiply(int[][] m1, int[][] m2) {
    int m1rows = m1.length;
    int m1cols = m1[0].length;
    int m2rows = m2.length;
    int m2cols = m2[0].length;
    if (m1cols != m2rows)
      throw new IllegalArgumentException(
        "matrices don't match: " + m1cols + " != " + m2rows);
    int[][] result = new int[m1rows][m2cols];

    // multiply
    for (int i=0; i<m1rows; i++) {
      for (int j=0; j<m2cols; j++) {
        for (int k=0; k<m1cols; k++) {
          result[i][j] += m1[i][k] * m2[k][j];
        }
      }
    }

    return result;
  }

  /** Matrix print.
   */
  public static void mprint(int[][] a) {
    int rows = a.length;
    int cols = a[0].length;
    System.out.println("array["+rows+"]["+cols+"] = {");
    for (int i=0; i<rows; i++) {
      System.out.print("{");
      for (int j=0; j<cols; j++)
        System.out.print(" " + a[i][j] + ",");
      System.out.println("},");
    }
    System.out.println("};");
```

```
  }
}
```

Here is a program (in the same source directory) that uses the `Matrix` class to multiply two arrays of `int`s:

```
int x[][] = {
  { 3, 2, 3 },
  { 5, 9, 8 },
};
int y[][] = {
  { 4, 7 },
  { 9, 3 },
  { 8, 1 },
};
int z[][] = Matrix.multiply(x, y);
Matrix.mprint(x);
Matrix.mprint(y);
Matrix.mprint(z);
```

See Also

Consult a book on numerical methods for more things to do with matrices; one of our reviewers recommends the series of *Numerical Recipes* books by William Press et al. (Cambridge University Press), available from the Numerical Recipes website. There are several translations of the book's code into various languages, including Java. Pricing varies by package; the book and the accompanying software are priced separately.

There are some commercial software packages that can do some of these calculations for you. Do a web search to find the current offerings.

Note: for single-dimension arrays (e.g., vectors), you can just use the Vector API described in Recipe 5.11.

5.11 Optimizing Large Arithmetic Operations with Vector Operations 22C

Problem

You need to perform arithmetic operations on a large collection of numbers.

Solution

Use the Vector arithmetic package.

Discussion

Most modern computer processors (CPUs) include machine instructions for performing vector calculations in hardware, faster than one could perform by writing code in software. These instructions will operate on two vectors (*arrays* in Java terms), and take a set of numbers (usually four, eight, or some multiple) of numbers from each, and perform the given operation (add, multiply, etc.) on all eight numbers from each of the two vectors, *in a single machine cycle*, thus significantly speeding things up. This is called Single Instruction Multiple Data, or SIMD, and it is one of the *S* letters in Streaming SIMD Extensions (SSE). SSE is one of several vector instruction sets, including:

- SSE and Advanced Vector Extensions (AVX) on amd64/Intel 64-bit architectures
- Neon and Scalable Vector Extensions (SVE) on ARM AArch64 architectures

Naturally, chip makers such as AMD and Intel try to be compatible with each other while extending the CPU features to try to out-do each other. Java's vector operations support tries to do the best it can to take advantage of these features, while providing equally accurate software implementation of operations that are not supported on a given machine.

The Vector API is in incubation status as of Java 22, so you need to pass the `--add-module=jdk.incubator.vector` and import from that package when using this API in software.

While Java already has some auto-vectorization, it is hard to predict when it will kick in. The Vector API makes the vectorization explicit, and is intended to work on any platform, using SIMD instructions if available and falling back to a software implementation if SIMD is not available.

There is a series of specialized terminology used here. Elements within a Vector are called *lanes*, as they can run in parallel, like the lanes of a highway. The *shape* of a Vector is how many bits it can process at a time. `Vector<E>` is abstract; there exists a concrete Vector subclass for each of Java's six numeric types (excluding `char`): `Byte Vector`, `ShortVector`, `IntVector`, `LongVector`, `FloatVector`, and `DoubleVector`. The combination of data type and shape is called a *species*.

In using the API, one has to choose a *species* of vectorization, or just let the system choose by using one labeled `PREFERRED`. The species will usually be stored in a `static final` variable for performance reasons. The array data has to be wrapped in one of

the six Vector subclasses. Then the operations are specified, using method names rather than syntax operators.

The code in Example 5-15 shows an example of a simple vector calculation of the formula $ax^2 + 2b$ on each pair of numbers taken from two parallel arrays of `doubles`. The calculation is done on a small array of numbers, both using the Vector API and performing the (reasonably simple) calculation explicitly.

Example 5-15. incubation/src/main/java/numbers/VectorOps.java—Demo of Vector API

```
import jdk.incubator.vector.DoubleVector;
import jdk.incubator.vector.VectorSpecies;

/** Compute ax^2+2b on double arrays */
public class VectorOps {

  static final VectorSpecies<Double> SPECIES = DoubleVector.SPECIES_PREFERRED;

  public static void main(String[] args) {
    System.out.println("Using Vector API with Species " + SPECIES);
    double[] a = { 2, 4, 6, 8, 9, 10, 11, 12, 13, 14};
    double x = Math.PI;
    double[] b = { 1, 3, 5, 7, 9, 10, 11, 12, 13, 14};
    double[] sresults = new double[a.length], vresults = new double[a.length];
    System.out.println(
      "Inputs: a=" + Arrays.toString(a) +
        ", x = " + x + ", b = " + Arrays.toString(b));

    scalarComputation(a, x, b, sresults);
    System.out.println("Results from scalar = " +
      Arrays.toString(sresults) + ")");

    vectorComputation(a, x, b, vresults);
    System.out.println("Results from vector = " +
      Arrays.toString(vresults) + ")");

    for (int i = 0; i < sresults.length; i++) {
      if (Math.abs(sresults[i] - vresults[i]) > 0.00000001D) {
        throw new IllegalStateException(
          "Values differ (" +
          sresults[i] + " != " + vresults[i] + ")");
      }
    }
    System.out.println("Computed both ways: Close enough!");
  }

  static void scalarComputation(double[] a, double x, double[] b, double[] c) {
    for (int i = 0; i < a.length; i++) {
      c[i] = (a[i] * x * x + b[i] * 2);
    }
```

```
  }

  static void vectorComputation(double[] a, double x, double[] b, double[] c) {
    int i = 0;
    int upperBound = SPECIES.loopBound(a.length);
    for (; i < upperBound; i += SPECIES.length()) {
      var va = DoubleVector.fromArray(SPECIES, a, i);
      var vb = DoubleVector.fromArray(SPECIES, b, i);
      System.out.println("va length = " + va.length() + ")");
      var vc = va
          .mul(x*x)
          .add(vb.mul(2));
      vc.intoArray(c, i);
    }
    // Handle leftover (iff input length % SPECIES.length() != 0)
    for (; i < a.length; i++) {
      c[i] = (a[i] * x * x + b[i] * 2);
    }
  }
}
```

The simplistic implementation is clearly shorter, but the Vector implementation will be faster, though it's not easy to measure the difference on such a small dataset. The spread will almost certainly increase as the dataset size increases. The answers are almost the same, except that one differs in the 14th digit (compare the last digit of the sixth number in each result). This tiny difference shows that the Vector and software solutions provide the "same" answer, while confirming that different code was used to produce the two outputs.

Running the program produces the following output. The `scalar` results are from the simplistic implementation, and the `vector` results are from the Vector API:

```
$ java --add-modules jdk.incubator.vector \
incubation/src/main/java/numbers/VectorOps.java
WARNING: Using incubator modules: jdk.incubator.vector
Note: ../incubation/src/main/java/numbers/VectorOps.java uses preview features
of Java SE 21.
Note: Recompile with -Xlint:preview for details.
Using Vector API with Species Species[double, 4, S_256_BIT]
Inputs: a=[2.0, 4.0, 6.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0], x =
3.141592653589793, b = [1.0, 3.0, 5.0, 7.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0]
Results from scalar = [21.739208802178716, 45.47841760435743,
69.21762640653614, 92.95683520871486, 106.82643960980423, 118.69604401089357,
130.56564841198292, 142.43525281307228, 154.30485721416167, 166.174461615251]
va length = 4
va length = 4
Results from vector = [21.739208802178716, 45.47841760435743,
69.21762640653614, 92.95683520871486, 106.82643960980423, 118.69604401089359,
130.56564841198292, 142.43525281307228, 154.30485721416167, 166.174461615251]
```

```
Computed both ways: Close enough!
$
```

As can be seen from the species printout, this species of computation takes four elements at a time. Thus the printout "va length = 4" occurs twice, taking 8 lanes of the 10 in the vector. The remaining two are handled by the extra loop at the end of `vectorComputation()`.

The `Vector` class features quite a few other operators. The basic arithmetic ones are `add()`, `sub()`, `mul()`, `div()`, `abs()`, `min()`, `max()`, `neg()`, `eq()`, `lt()` (but no `gt()`). There are many, many methods for reshaping, extracting, and slicing and dicing the vectors. The best reference is the Javadoc page for the `Vector` class. Although it's in incubation, this man page is included in the standard Javadoc set.

5.12 Using Complex Numbers

Problem

You need to manipulate complex numbers, as is common in mathematical, scientific, or engineering applications.

Solution

Find, or write, a class to keep track of the real and imaginary parts of a complex number.

Discussion

Java does not provide any explicit support for dealing with complex numbers. You could keep track of the real and imaginary parts and do the computations yourself, but that is not a very well-structured solution.

A better solution, of course, is to use a class that implements complex numbers. I have written such a class, but now I recommend using the `Complex` class in the Apache Commons Math library, which is more complete. The build coordinates for this are `org.apache.commons:commons-math3:3.6.1` (or later). First, an example of using Apache's library:

```
public class ComplexDemoACM {

  public static void main(String[] args) {
    Complex c = new Complex(3,  5);
    Complex d = new Complex(2, -2);
    System.out.println(c);
    System.out.println(c + ".getReal() = " + c.getReal());
    System.out.println(c + " + " + d + " = " + c.add(d));
    System.out.println(c + " * " + d + " = " + c.multiply(d));
```

```
        System.out.println(c.divide(d));
      }
    }
```

Running this demo program produces the following output:

```
(3.0, 5.0)
(3.0, 5.0).getReal() = 3.0
(3.0, 5.0) + (2.0, -2.0) = (5.0, 3.0)
(3.0, 5.0) * (2.0, -2.0) = (16.0, 4.0)
(-0.5, 2.0)
```

Example 5-16 is the source for my version of the `Complex` class and shouldn't require much explanation. The Apache class is admittedly more sophisticated, but I leave mine here just to demystify the basic operation of complex numbers.

To keep the API general, I provide—for each of add, subtract, and multiply—both a static method that works on two complex objects and a nonstatic method that applies the operation to the given object and one other object.

Example 5-16. main/src/main/java/numbers/Complex.java

```
public record Complex(double r, double i) {

  /** Display the current Complex as a String, for use in
   * println() and elsewhere.
   */
  public String toString() {
    StringBuilder sb = new StringBuilder().append(r);
    if (i>0)
      sb.append('+');  // else append(i) appends - sign
    return sb.append(i).append('i').toString();
  }

  /** Return just the Real part */
  public double getReal() {
    return r;
  }
  /** Return just the Real part */
  public double getImaginary() {
    return i;
  }
  /** Return the magnitude of a complex number */
  public double magnitude() {
    return Math.sqrt(r*r + i*i);
  }

  /** Add another Complex to this one
   */
  public Complex add(Complex other) {
    return add(this, other);
  }
```

```
  /** Add two Complexes
   */
  public static Complex add(Complex c1, Complex c2) {
    return new Complex(c1.r+c2.r, c1.i+c2.i);
  }

  /** Subtract another Complex from this one
   */
  public Complex subtract(Complex other) {
    return subtract(this, other);
  }

  /** Subtract two Complexes
   */
  public static Complex subtract(Complex c1, Complex c2) {
    return new Complex(c1.r-c2.r, c1.i-c2.i);
  }

  /** Multiply this Complex times another one
   */
  public Complex multiply(Complex other) {
    return multiply(this, other);
  }

  /** Multiply two Complexes
   */
  public static Complex multiply(Complex c1, Complex c2) {
    return new Complex(c1.r*c2.r - c1.i*c2.i, c1.r*c2.i + c1.i*c2.r);
  }

  /** Divide c1 by c2.
   * @author Gisbert Selke.
   */
  public static Complex divide(Complex c1, Complex c2) {
    return new Complex(
      (c1.r*c2.r+c1.i*c2.i)/(c2.r*c2.r+c2.i*c2.i),
      (c1.i*c2.r-c1.r*c2.i)/(c2.r*c2.r+c2.i*c2.i));
  }
}
```

Running the simple demo file `ComplexDemoIan` in the same directory will show this in action:

```
    3.0+5.0i
    3.0+5.0i.getReal() = 3.0
    3.0+5.0i + 2.0-2.0i = 5.0+3.0i
    3.0+5.0i + 2.0-2.0i = 5.0+3.0i
    3.0+5.0i * 2.0-2.0i = 16.0+4.0i
    3.0+5.0i / 2.0-2.0i = -0.5+2.0i
```

5.13 Handling Very Large Numbers

Problem

You need to handle integer numbers larger than `Long.MAX_VALUE` or floating-point values larger than `Double.MAX_VALUE`.

Solution

Use the `BigInteger` or `BigDecimal` values in package java.math, as shown in Example 5-17.

Example 5-17. main/src/main/java/numbers/BigNums.java

```
System.out.println("Here's Long.MAX_VALUE: " + Long.MAX_VALUE);
BigInteger bInt = new BigInteger("34192292233720368547758 07");
System.out.println("Here's a bigger number: " + bInt);
System.out.println("Here it is as a double: " + bInt.doubleValue());
```

Note that the constructor takes the number as a string. Obviously you couldn't just type the numeric digits because, by definition, these classes are designed to represent numbers larger than will fit in a Java `long`.

Discussion

Both `BigInteger` and `BigDecimal` objects are immutable; that is, once constructed, they always represent a given number. That said, a number of methods return new objects that are mutations of the original, such as `negate()`, which returns the negative of the given `BigInteger` or `BigDecimal`. There are also methods corresponding to most of the Java language built-in operators defined on the base types `int`/`long` and `float`/`double`. The division method makes you specify the rounding method; consult a book on numerical analysis for details. Example 5-18 is a simple stack-based calculator using `BigDecimal` as its numeric data type.

Example 5-18. main/src/main/java/numbers/BigNumCalc.java

```
public class BigNumCalc {

  /** an array of Objects, simulating user input */
  public static Object[] testInput = {
    new BigDecimal("34192292233720368547758 07.23343"),
    new BigDecimal("2.0"),
    "*",
  };

  public static void main(String[] args) {
```

```
    BigNumCalc calc = new BigNumCalc();
    System.out.println(calc.calculate(testInput));
  }

  /**
   * Stack of numbers being used in the calculator.
   */
  Stack<BigDecimal> stack = new Stack<>();

  /**
   * Calculate a set of operands; the input is an Object array containing
   * either BigDecimal objects (which may be pushed onto the Stack) and
   * operators (which are operated on immediately).
   * @param input
   * @return
   */
  public BigDecimal calculate(Object[] input) {
    BigDecimal tmp;
    for (int i = 0; i < input.length; i++) {
      Object o = input[i];
      if (o instanceof BigDecimal decimal) {
        stack.push(decimal);
      } else if (o instanceof String string) {
        switch (string.charAt(0)) {
        // + and * are commutative, order doesn't matter
        case '+':
          stack.push((stack.pop()).add(stack.pop()));
          break;
        case '*':
          stack.push((stack.pop()).multiply(stack.pop()));
          break;
        // - and /, order *does* matter
        case '-':
          tmp = stack.pop();
          stack.push((stack.pop()).subtract(tmp));
          break;
        case '/':
          tmp = stack.pop();
          stack.push((stack.pop()).divide(tmp, RoundingMode.HALF_UP));
          break;
        default:
          throw new IllegalStateException("Unknown OPERATOR popped");
        }
      } else {
        throw new IllegalArgumentException("Syntax error in input");
      }
    }
    return stack.pop();
  }
}
```

Running this produces the expected (very large) value:

```
$ java numbers/BigNumCalc.java
6838458446744073709551614.466860
$
```

The current version has its inputs hardcoded, as does the JUnit test program, but in real life you can use regular expressions to extract words or operators from an input stream (as in Recipe 4.6), or you can use the `StreamTokenizer` approach of the simple calculator (see Recipe 10.7). The stack of numbers is maintained using a `java.util.Stack` (see Recipe 7.6).

`BigInteger` is mainly useful in cryptographic and security applications. Its method `isProbablyPrime()` can create prime pairs for public key cryptography. `BigDecimal` might also be useful in computing the size of the universe.

5.14 Program: TempConverter

The program shown in Example 5-19 prints a table of Fahrenheit temperatures (still used in daily weather reporting in a few countries like the US and its territories, Liberia, and some countries in the Caribbean) and the corresponding Celsius temperatures (used in science everywhere and in daily life in most of the world).

Example 5-19. main/src/main/java/numbers/TempConverter.java

```
public class TempConverter {

  public static void main(String[] args) {
    TempConverter t = new TempConverter();
    t.start();
    t.data();
    t.end();
  }

  protected void start() {
  }

  protected void data() {
    for (int i=-40; i<=120; i+=10) {
      double c = fToC(i);
      print(i, c);
    }
  }

  public static double cToF(double deg) {
    return ( deg * 9 / 5) + 32;
  }

  public static double fToC(double deg) {
```

```
    return ( deg - 32 ) * ( 5d / 9 );
  }

  protected void print(double f, double c) {
    System.out.println(f + " " + c);
  }

  protected void end() {
  }
}
```

This works, but these numbers print with about 15 digits of (useless) decimal fractions! The second version of this program subclasses the first and uses `printf` (see Recipe 3.2) to control the formatting of the converted temperatures (see Example 5-20). It will now look much better, assuming you're printing in a monospaced font.

Example 5-20. main/src/main/java/numbers/TempConverter2.java

```
public class TempConverter2 extends TempConverter {

  public static final String SEP = "-----------------";

  public static void main(String[] args) {
    TempConverter t = new TempConverter2();
    t.start();
    t.data();
    t.end();
  }

  @Override
  protected void print(double f, double c) {
    System.out.printf("%6.2f\u00B0F %6.2f\u00B0C%n", f, c);
  }

  @Override
  protected void start() {
    System.out.println("Fahr      Celsius");
    System.out.println(SEP);
  }

  @Override
  protected void end() {
    System.out.println(SEP);
  }
}
```

Here is the output of running this program:

```
$ java numbers.TempConverter2
Fahr      Celsius
-----------------
-40.00°F -40.00°C
-30.00°F -34.44°C
-20.00°F -28.89°C
-10.00°F -23.33°C
  0.00°F -17.78°C
 10.00°F -12.22°C
 20.00°F  -6.67°C
 30.00°F  -1.11°C
 40.00°F   4.44°C
 50.00°F  10.00°C
 60.00°F  15.56°C
 70.00°F  21.11°C
 80.00°F  26.67°C
 90.00°F  32.22°C
100.00°F  37.78°C
110.00°F  43.33°C
120.00°F  48.89°C
-----------------
$
```

CHAPTER 6

Dates and Times

6.0 Introduction

Developers suffered for a decade and a half under the inconsistencies and ambiguities of the `Date` class from Java 1.0 and its replacement wannabe, the `Calendar` class from Java 1.1. Several alternative `Date` replacement packages emerged, including the simple and sensible Date4J and the more comprehensive Joda-Time package. Java 8 introduced a new, consistent, and well-thought-out package for date and time handling under the aegis of the Java Community Process, JSR-310,[1] shepherded by developer Stephen Colebourne. Stephen says that "JSR-310 started from scratch, but with an API 'inspired by Joda-Time,'" his earlier package, with several important design changes.[2] This package is biased toward ISO 8601 dates; the default format is, for example, 2015-10-23T10:22:45. But it can, of course, work with other calendar schemes.

One of the key benefits of the new API is that it provides *useful operations* such as adding/subtracting dates/times. Much time was wasted by developers reimplementing these useful operations for the old APIs. With the new APIs, one can use the built-in functionality. That said, millions of lines of code are based on the old APIs, so we'll review them briefly and then consider interfacing the new API to legacy code, in the final recipe of this chapter, Recipe 6.9.

1 JSR stands for Java Specification Request. The Java Community Process calls standards, both proposed and adopted, JSRs. See the Java Community Process for details.

2 For those with an interest in historical arcana, the differences are documented on his blog.

Another advantage of the new API is that almost all objects are immutable and thus thread-safe. This can be of considerable benefit as we move headlong into the massively parallel era.

Because there are no `set` methods, and thus the getter method paradigm doesn't always make sense, the API provides a series of new methods to replace such methods, listed in Table 6-1.

Table 6-1. New date/time API: common methods

Name	Description
`at`	Combines with another object
`format`	Use provided formatter to produce a formatted string
`from`	Factory: convert input parameters to instance of target
`get`	Retrieve one field from the instance
`is...`	Examine the state of the given object
`minus`	Return a copy with the given amount subtracted
`now`	BuilderFactory: get the current time, date, etc.
`of`	Factory: create new method by parsing inputs
`parse`	Factory: parse single input string to produce instance of target
`plus`	Return a copy with the given amount added
`to`	Return a copy of this object converted to another type
`with`	Return a copy with the given field changed; replaces `set` methods

The JSR-310 API specifies a dozen or so main classes. Those representing time are either continuous time or human time. *Continuous time* is based on Unix time, a deeper truth from the dawn of (computer) time, and is represented as a single monotonically increasing number. The time value of 0 in Unix represents the first second of January 1, 1970 UTC—about the time Unix was invented.[3] Each unit of increment thereafter represents one second of time. This has been used as a time base in most operating systems developed since. However, a 32-bit integer representing the number of seconds since 1970 runs out fairly soon—in the year 2038 AD. Most Unix systems have, in the aftermath of the Y2K frenzy, quietly and well in advance, headed off the possibility of a Y2038 frenzy by converting the time value from a 32-bit quantity to a 64-bit quantity.

Java also used this time base, but used 64 bits and stored its time in milliseconds, because a 64-bit time in milliseconds since 1970 will not overflow until quite a few

3 This is late in the day of December 31, 1969, in the Eastern time zone, so when you run across a file with a "last modified" time in that day, it pretty much invariably means that part of the file's metadata got set to zero by accident.

years into the future (keep this date open in your calendar—August 17, 292,278,994 AD). Here is a calculation that shows how I got that date:

```
Date endOfTime = new Date(Long.MAX_VALUE);
System.out.println("Java8 time overflows on " + endOfTime);
```

The new date/time API is in five packages, as shown in Table 6-2; as usual, the top-level package contains the most commonly used pieces.

Table 6-2. New date/time API: packages

Name	Description
java.time	Common classes for dates, times, instants, and durations
java.time.chrono	API for non-ISO calendar systems
java.time.format	Formatting classes (see Recipe 6.2)
java.time.temporal	Date and time access using fields, units, and adjusters
java.time.zone	Support for time zones and their rules

The basic java.time package contains a dozen or so classes, as well as a couple of enums and one general-purpose exception (shown in Tables 6-3, 6-4, and 6-5).

Table 6-3. New date/time API: basics

Class	Description
`Clock`	Replaceable factory for getting current time
`Instant`	A point in time since January 1, 1970, expressed in nanoseconds
`Duration`	A length of time, also expressed in nanoseconds

Human time represents times and dates as we use them in our everyday life. These classes are listed in Table 6-4.

Table 6-4. New date/time API: human time

Class	Description
`DayOfWeek`	A day of the week (e.g., Tuesday)
`LocalDate`	A bare date (day, month, and year) with no adjustments
`LocalTime`	A bare time (hour, minute, seconds) with no adjustments
`LocalDateTime`	The combination of the preceding classes
`Month`	Month-of-year (e.g., January)
`MonthDay`	Month and day (e.g., 02-29)
`OffsetTime`	A time of day with a time zone offset like –04:00, with no date or zone
`OffsetDateTime`	A date and time with a time zone offset like –04:00, with no time zone
`Period`	A descriptive amount of time, such as "2 months and 3 days," stored in a compact string

Class	Description
`Year`	A year by itself
`YearMonth`	A year and month
`ZonedDateTime`	The date and time with a time zone and an offset

Almost all the top-level classes directly extend `java.lang.Object` and are held to consistency by a variety of interfaces, which are declared in the subpackages. The date and time classes mostly implement `Comparable`, which makes sense.

Table 6-5 shows the two time-zone-specific classes used with `ZonedDateTime`, `OffsetDateTime`, and `OffsetTime`.

Table 6-5. New date/time API: support

Class	Description
`ZoneId`	Defines a time zone such as *Canada/Eastern* and its conversion rules
`ZoneOffset`	A time offset from UTC (hours, minutes, seconds)

The new API is a *fluent API*, in which most operations return the object they have operated upon, so that you can chain multiple calls without the need for tedious and annoying temporary variables:

```
LocalTime time = LocalTime.now().minusHours(5); // the time 5 hours ago
```

This results in a more natural and convenient coding style, in my opinion. You can always write code with lots of temporary variables if you want; you're the one who will have to read through it later.

6.1 Finding Today's Date

Problem

You want to find today's date and/or time.

Solution

Invoke the appropriate builder to obtain a `LocalDate`, `LocalTime`, or `LocalDateTime` object. Call its `toString()` method if you want it as a `String`, e.g., to display.

Discussion

These classes do not provide public constructors, so you will need to call one of the factory methods to get an instance. These classes each provide a `now()` method,

which does what its name implies. The `CurrentDateTime` demo program shows simple use of all three:

```
public class CurrentDateTime {
  public static void main(String[] args) {
    LocalDate dNow = LocalDate.now();
    System.out.println(dNow);
    LocalTime tNow = LocalTime.now();
    System.out.println(tNow);
    LocalDateTime now = LocalDateTime.now();
    System.out.println(now);
  }
}
```

Running it produces this output:

```
2013-10-28
22:23:55.641
2013-10-28T22:23:55.642
```

The formatting is nothing spectacular, but it's adequate. Fancier formatting is dealt with in Recipe 6.2.

While this works, in full-scale applications, it's recommended to pass a `Clock` instance into all the `now()` methods, to make testing easier. `Clock` is a factory object that is used internally to find the current time. In testing, you often want to have a known date or time used so you can compare against known output. The `Clock` class makes this easy. Example 6-1 uses a `Clock` and allows replacing the default `Clock` by calling a setter. Alternatively you could use a dependency injection framework like CDI or Spring to provide the correct version of the `Clock` class.

Example 6-1. main/src/main/java/datetime/TestableDateTime.java

```
/**
 * TestableDateTime allows test code to plug in a fixed-value
 * clock for e.g., unit testing.
 */
public class TestableDateTime {
  private static Clock clock = Clock.systemDefaultZone();
  public static void main(String[] args) {
    System.out.println("It is now " + LocalDateTime.now(clock));
  }
  public static void setClock(Clock clock) {
    TestableDateTime.clock = clock;
  }
}
```

In normal operation this would get the current date and time. In testing you would call the `setClock()` method with a `Clock` instance obtained from the static method `Clock.fixed(Instant fixedInstant, ZoneId zone)`, passing in the time that your

testing code expects. The fixed clock does not tick, so don't worry about the milliseconds between setting the clock to fixed and the invocation of your tests.

6.2 Formatting Dates and Times

Problem

You want to provide better formatting for date and time objects.

Solution

Use `java.time.format.DateTimeFormatter`.

Discussion

The `DateTimeFormatter` class allows you to format dates and times. Its `format()` methods will accept any `TemporalAccessor`, which includes `LocalDate`, `LocalDateTime`, `LocalTime`, `ZonedDateTime`, and a dozen others, including date representations from other calendar styles than the ISO 8601 calendar. There are three categories of methods for getting a `DateTimeFormatter` instance from the class:

- Predefined formatter instances
- Localized formatter instances
- Pattern-based formatting

Predefined formatter instances

There are quite a few predefined instances, as shown in Table 6-6.

Table 6-6. Predefined `DateTimeFormatter` instances

Name	Meaning	Example
BASIC_ISO_DATE	Basic ISO date	*20301111*
ISO_LOCAL_DATE	ISO Local Date	*2030-11-11*
ISO_OFFSET_DATE	ISO Date with offset	*2030-11-11+11:00*
ISO_DATE	ISO Date with or without offset	*2030-11-11+01:00; 2030-11-11*
ISO_LOCAL_TIME	Time without offset	*11:11:30*
ISO_OFFSET_TIME	Time with offset	*11:11:30+01:00*
ISO_TIME	Time with or without offset	*11:11:30+01:00; 11:11:00*
ISO_LOCAL_DATE_TIME	ISO Local Date and Time	*2030-11-13T11:11:00*
ISO_OFFSET_DATE_TIME	Date Time with Offset	*2030-12-03T11:11:00+01:00*
ISO_ZONED_DATE_TIME	Zoned Date Time	*2030-12-03T11:11:00+01:00[Europe/Paris]*

Name	Meaning	Example
ISO_DATE_TIME	Date and time with ZoneId	*2030-12-03T11:11:00+01:00[Europe/Paris]*
ISO_ORDINAL_DATE	Year and day of year	*2012-337*
ISO_WEEK_DATE	Year and Week	*2012-W48-6*
ISO_INSTANT	Date and Time of an Instant	*2030-12-03T11:11:00Z*
RFC_1123_DATE_TIME	RFC 1123/RFC 822	*Tue, 3 Jun 2030 11:11:00 GMT*

These can all be used as formatters, as in this example:

```
System.out.println(
        DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(
                LocalDateTime.now()));
2024-10-16T14:33:57.810557801
```

Localized DateTimeFormatter instances

You can get a localized formatter with one of the following static methods. If you are changing the default `Locale`, do so before calling these methods.

`ofLocalizedDate(dateStyle)`
: Get a formatter with date style from the locale, e.g., *2030-11-11*

`ofLocalizedTime(timeStyle)`
: Get a formatter with time style from the locale, e.g., *11:11:00*

`ofLocalizedDateTime(dateTimeStyle)`
: Get a formatter with a style for date and time from the locale, e.g., *3 Jun 2008 11:05:30*

`ofLocalizedDateTime(dateStyle,timeStyle)`
: Get a formatter with date and time styles from the locale, e.g., *3 Jun 2008 11:05*

These take one or two `style` arguments from the `FormatStyle` enum, which has four values: `FULL`, `LONG`, `MEDIUM`, and `SHORT`. Some examples of this are in Example 6-2.

Example 6-2. main/src/main/java/datetime/DateTimeFormatterLocalized.java

```
public class DateTimeFormatterLocalized {

  public static void main(String[] args) {

    var dt = ZonedDateTime.now();

    for (Locale l :
      List.of(Locale.CANADA, Locale.FRANCE, Locale.UK, Locale.TAIWAN)) {
      Locale.setDefault(l);
      DateTimeFormatter f =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
```

```
      System.out.printf(
        "In Locale %s, date is %s\n",Locale.getDefault(), f.format(dt));
    }
  }
}
```

Running this shows the current date and time in several different locale settings:

```
In Locale en_CA, date is Oct 16, 2024, 5:25:14 p.m.
In Locale fr_FR, date is 16 oct. 2024, 17:25:14
In Locale en_GB, date is 16 Oct 2024, 17:25:14
In Locale zh_TW, date is 2024 年 10 月 16 日 下午 5:25:14
```

The Taiwan output is only garbled because of font issues.

Any `DateTimeFormatter` instance can also be (re)localized (see Recipe 3.12) after construction. Whether you used one of the `ofLocalized…()` factory methods or not, you can always get an instance configured for a new locale by calling `withLocale()` on the instance; as the name hints, you get back a new instance (remember that all classes in `java.time` create immutable objects):

```
void main() {
  var df = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
  var now = LocalDateTime.now();
  System.out.println(df.format(now));
  var newdf = df.withLocale(Locale.UK); // new instance with updated Locale info
  System.out.println(newdf.format(now));
}
```

The output shows a given date and time formatted in my default locale and in the UK locale:

```
Oct 16, 2024, 5:50:00 PM
16 Oct 2024, 17:50:00
```

Pattern-based formatting

Finally, if you need the ultimate in flexibility or just don't want to use one of the provided predefined formats, you can define your own by using `DateTimeFormatter.ofPattern(String pattern)`. The `pattern` string can contain any characters, but almost every letter of the alphabet has been defined to mean something, in addition to the obvious `Y`, `M`, `D`, `h`, `m`, and `s`. In addition, the quote character and square bracket characters are defined, and all letters in the Roman alphabet as well as the sharp sign (`#`) and curly braces are reserved.

As is common with date formatting languages, the number of repetitions of a letter in the pattern gives a clue to its intended length of detail. Thus, for example, `MMM` gives `Jan`, whereas `MMMM` gives `January`.

Don't confuse this with `java.text.DateFormat`, which works on the obsolete `java.util.Date` class.

Table 6-7 is an attempt at a complete list of the formatting characters, adapted from the Javadoc for the `DateTimeFormatter` class.

Table 6-7. `DateTimeFormatter` format characters

Symbol	Meaning	Presentation	Examples
G	Era	Text	AD; Anno Domini
y	Year of era	Year	2004; 04
u	Year of era	Year	See the following note.
D	Day of year	Number	189
M/L	Month of year	Number/text	7; 07; Jul; July; J
d	Day of month	Number	10
Q/q	Quarter of year	Number/text	3; 03; Q3, 3rd quarter
Y	Week-based year	Year	1996; 96
w	Week of week-based year	Number	27
W	Week of month	Number	4
e/c	Localized day of week	Number/text	2; 02; Tue; Tuesday; T
E	Day of week	Text	Tue; Tuesday; T
F	Week of month	Number	3
a	a.m./p.m. of day	Text	PM
h	Clock hour of a.m./p.m. (1-12)	Number	12
K	Hour of a.m./p.m. (0-11)	Number	0
k	Clock hour of a.m./p.m. (1-24)	Number	0
H	Hour of day (0-23)	Number	0
m	Minute of hour	Number	30
s	Second of minute	Number	55
S	Fraction of second	Fraction	978
A	Millisecond of day	Number	1234
n	Nanosecond of second	Number	987654321
N	Nanosecond of day	Number	1234000000
V	Time zone ID	Zone-id	America/Los_Angeles; Z; –08:30
z	Time zone name	Zone-name	Pacific Standard Time; PST
X	Zone offset *Z* for zero	Offset-X	Z; –08; –0830; –08:30; –083015; –08:30:15;
x	Zone offset	Offset-x	+0000; –08; –0830; –08:30; –083015; –08:30:15;

Symbol	Meaning	Presentation	Examples
z	Zone offset	Offset-Z	+0000; –0800; –08:00;
o	Localized zone offset	Offset-O	GMT+8; GMT+08:00; UTC–08:00;
p	Pad next	Pad modifier	1

Unlike `printf`-style formatting, do not precede these format specifiers with the `%` symbol.

Some examples are shown in Example 6-3.

Example 6-3. main/src/main/java/datetime/DateTimeFormatterDemo.java

```
public class DateTimeFormatterDemo {
  public static void main(String[] args) {

    // Format a date ISO8601-like but with slashes instead of dashes
    DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy/LL/dd");
    System.out.println(df.format(LocalDate.now()));

    // Format a ZonedDateTime without timezone information
    DateTimeFormatter nTZ =
      DateTimeFormatter.ofPattern("d MMMM, yyyy h:mm a");
    System.out.println(ZonedDateTime.now().format(nTZ));
  }
}
```

Where there are two letters with a slash between, the first is the "standard" form and the second is the "standalone" form, e.g, when displaying that field by itself rather than as part of a date/time string. With one or two copies of the character, it prints numerically, with a leading zero if needed. With three, it prints a short form, with four, a longer form. With five, it prints the "abbreviated form," which is the shortest possible, e.g., `O` for October. In practice there seems little difference, as shown by Example 6-4.

Example 6-4. main/src/main/java/datetime/DateTimeFormatterLM.java

```
/// Try to show the difference between "standard" and "standalone"
/// formats such as "M/L" in [java.time.format.DateTimeFormatter].
///
public class DateTimeFormatterLM {
  public static void main(String[] args) {
    System.out.println("L       M");
    var then = LocalDate.of(2222,2,2);
    IntStream.rangeClosed(1, 5).forEach( i -> {
```

```
        var str = "L".repeat(i) + " " + "M".repeat(i);
        var fmt = DateTimeFormatter.ofPattern(str);
        System.out.printf("Length %d (%s) gives %s\n", i, str, fmt.format(then));
    });
  }
}
```

Running this program prints the following, showing all five formats:

```
$ java src/main/java/datetime/DateTimeFormatterLM.java
L       M
Length 1 (L M) gives 2 2
Length 2 (LL MM) gives 02 02
Length 3 (LLL MMM) gives Feb Feb
Length 4 (LLLL MMMM) gives February February
Length 5 (LLLLL MMMMM) gives F F
```

y and u work the same for AD years; however, for a year of 3 BC, the y pattern returns 3, whereas the u pattern returns –2 (a.k.a. proleptic year).

6.3 Converting Among Dates/Times and Epoch Seconds

Problem

You need to convert among dates/times, epoch seconds, or some other numeric value.

Solution

Use the appropriate date/time factory or retrieval methods.

Discussion

The epoch is the beginning of time as far as modern operating systems go. Unix time, and some versions of Windows time, count off inexorably the seconds since the epoch. When Ken Thompson and Dennis Ritchie came up with this format in 1970, seconds seemed like a fine measure, and 32 bits' worth of seconds seemed nearly infinite. On operating systems that store the epoch in a 32-bit integer, however, time is running out. Older versions of most operating systems stored this as a 32-bit signed integer, which unfortunately will overflow on January 18, 2038, at 22:14:07 Eastern time.

When Java first came out, it featured a method called `System.currentTimeMillis()`, which presented epoch seconds with millisecond accuracy.[4]

Any of these epoch-related numbers can be converted into, or obtained from, a local date/time. Other numbers can also be used, such as integer years, months, and days. As usual, there are factory methods that create new objects where a change is requested. Here is a program that shows some of these conversions in action:

main/src/main/java/datetime/DateConversions.java

```
// Convert "a billion seconds of Unix" to a local date/time
Instant epochSec = Instant.ofEpochSecond(1000000000L);
ZoneId zId = ZoneId.systemDefault();
ZonedDateTime then = ZonedDateTime.ofInstant(epochSec, zId);
System.out.println("The epoch was a billion seconds old on " + then);

// Convert a date/time to Epoch seconds
long epochSecond = ZonedDateTime.now().toInstant().getEpochSecond();
System.out.println("Current epoch seconds = " + epochSecond);

ZonedDateTime here = ZonedDateTime.now();
ZonedDateTime there = here.withZoneSameInstant(ZoneId.of("Canada/Pacific"));
System.out.printf("When it's %s here, it's %s in Vancouver%n", here, there);
```

6.4 Parsing Strings into Dates

Problem

You need to convert user input into `java.time` objects.

Solution

Use a `parse()` method.

Discussion

Many of the date/time classes have a `parse()` factory method, which tries to parse a string into an object of that class. For example, `LocalDate.parse(String)` returns a `LocalDate` object for the date given in the input `String`, as shown in Example 6-5.

4 The modern Java API uses a nanosecond-resolution timestamp that is *not* related to the epoch (i.e., can only be compared with the same value later in the same run of the JVM). It can be obtained with a call to `System.nanoTime()`.

Example 6-5. main/src/main/java/datetime/DateParse.java

```
public class DateParse {
  public static void main(String[] args) {

    String armisticeDateString = "1918-11-11";
    LocalDate armisticeDate = LocalDate.parse(armisticeDateString);
    System.out.println("Date: " + armisticeDate);

    String armisticeDateTimeString = "1918-11-11T11:00";
    LocalDateTime armisticeDateTime =
      LocalDateTime.parse(armisticeDateTimeString);
    System.out.println("Date/Time: " + armisticeDateTime);
```

The `parse()` routines in these classes will throw a `DateTimeParseException` if the input string cannot be parsed according to the (default or specified) pattern.

As you probably expect by now, the default format is the ISO 8601 date format. However, we often have to deal with dates in other formats. There is a series of predefined formatters defined as fields in the `DateTimeFormatter` class, such as `BASIC_ISO_DATE`, `ISO_LOCAL_DATE_TIME`, `ISO_LOCAL_TIME`, and more—see Table 6-1 and also the Javadoc for the complete list. Additionally, you might want to define your own specific format. For this, the `DateTimeFormatter` allows you to specify a particular pattern. For example, `"dd MMM uuuu"` represents the day of the month (two digits), three letters of the name of the month (Jan, Feb, Mar, …), and a four-digit year:

```
    // DateParse.java (continued)
    // You can also parse with any DateTimeFormatter that matches your input
    DateTimeFormatter df = DateTimeFormatter.ofPattern("dd MMM uuuu");
    String anotherDate = "27 Jan 2027";
    LocalDate random = LocalDate.parse(anotherDate, df);
    System.out.println(anotherDate + " parses as " + random);
```

The `DateTimeFormatter` object is bidirectional; it can both parse input and format output. We could add this line to the `DateParse` example:

```
// DateParse.java (continued)
System.out.println(armisticeDate + " formats as " + df.format(armisticeDate));
```

When we run the program, we see the output as follows:

```
Date: 1918-11-11
Date/Time: 1918-11-11T11:00
27 Jan 2027 parses as 2027-01-27
1918-11-11 formats as 11 Nov 1918
```

Instants, Durations, and Periods

Timekeeping is inherently complex, as we live on a large globe—the Earth—that spins at a rate of *roughly* 24 hours (86,400 seconds) per day. Minor adjustments have to be made from time to time, as this planetary day is somewhat irregular. If this is of interest, please refer to the Javadoc for `Instant`. When reading this and related documents, you will come across the uncommon word *proleptic Gregorian calendar*. This simply means the modern Gregorian/ISO 8601 calendar extrapolated into the past, ignoring details such as the *week and a half skipped* at many different times in different countries when converting from the Julian calendar.[5]

At least three classes represent particular points of interest. An `Instant` has about the same meaning as in everyday speech, a (very) specific point in time. A `Period` has one of the same meanings as in English, a number of days (represented as years, months, and days). A `Duration` is, as in the common tongue, a number of seconds (represented as hours, minutes, and seconds). `Period` and `Duration` therefore both represent *spans of time*. The `Instant` class represents a particular point in time, measured in seconds with nanosecond precision.

Both `Instant` and `Duration` (but not `Period`) store their time internally in two parts:

- A `long` representing the number of seconds since the beginning of (Unix/Java) time, January 1st, 1970, at 00:00 UTC
- An `int` representing the number of nanoseconds, which ranges from 0 to 999,999,999

Interestingly, parts of a period can be negative, and will be treated individually:

```
LocalDate.now();
$18 ==> 2024-05-04
jshell> LocalDate.now().plus(Period.of(4,-1,12));
$19 ==> 2028-04-16
```

The years field was added; the month was negative so it was subtracted, and the days value was positive so it was added. I'm not sure of the use case for this, but there it is.

A `Duration` represents a span of time that is usually a day or less. It is represented as a string in the form `PTnHnMn.sS` where each combination of a number *n* and the *following* character represents that number of hours, minutes, and seconds (with optional milli- and nanoseconds). It can be created using various factory and plus/minus methods:

5 We can blame the Romans for both calendars; the Julian was decreed by Roman Emperor Julius Caesar (before he failed to beware the ides of March in his own calendar system), and the Gregorian was decreed by Roman Catholic Pope Gregory XIII.

```
Duration.ofHours(12).plusHours(4).plusMillis(3).plusMinutes(6);
PT16H6M0.003S
```

There is a `plusNanos()` method but not a `plusMicros()`.

The `Period` class represents the difference between two instants. It's presented as a compact string in the form `PnYnMnDnHnMnS` where each combination of a number *n* and the following character represents that number of years, months, etc. For example, `P5Y2D` means five years and two days. While the `Period` class isn't meant to be human-friendly, there is a formatter for it in *darwinsys-api/src/main/java/com/darwinsys/formatting/DateFormatting.java* and a test program demonstrating its use.

The `Period` and `Duration` objects are similar, though obviously differing in scale. Another difference only shows up when they are used in certain calculations, e.g., when added to a `ZonedDateTime`. Adding a `Duration` would add an exact number of seconds, so that a `Duration` of one day is always exactly 24 hours. A `Period`, on the other hand, would add a "conceptual day," taking account of time differences such as that which occur when entering or leaving Daylight Saving Time.

I did say it was complicated.

The recipes in this section cover some uses of these classes.

6.5 Difference Between Two Dates

Problem

You need to compute the difference between two dates.

Solution

Use the static method `Period.between()` to find the difference between two `LocalDates`.

Discussion

Given two `LocalDate` objects, you can find the difference between them, as a `Period`, simply using the static `Period.between()` method. You can `toString()` the `Period` or, if its default format isn't good enough, format the result yourself:

```
import java.time.LocalDate;
import java.time.Period;

/**
 * Tutorial/Example of LocalDate date difference subtraction
 */
public class DateDiff {
```

```
  public static void main(String[] args) {
    /* The date at the end of the last century */
    LocalDate endOf20thCentury = LocalDate.of(2000, 12, 31);
    LocalDate now = LocalDate.now();
    if (now.getYear() > 2100) {
      System.out.println("The 21st century is over!");
      return;
    }

    Period diff = Period.between(endOf20thCentury, now);

    System.out.printf("The 21st century (up to %s) is %s old%n", now, diff);
    System.out.printf(
        "The 21st century is %d years, %d months and %d days old%n",
        diff.getYears(), diff.getMonths(), diff.getDays());
  }
}
```

I reran this code near the end of October 2024; the 20th century AD ended at the end of 2000 (not the end of 1999!), so the value should be about 23 $^{10}/_{12}$ years, and it was:

```
$ java datetime.DateDiff
The 21st century (up to 2024-10-21) is P23Y9M21D old
The 21st century is 23 years, 9 months and 21 days old
```

While the `Period` class isn't meant to be human-friendly, pulling out the fields as was done here is tedious.

Because of the API's regularity, you can use the same technique with `LocalTime` or `LocalDateTime`.

There is also `ChronoUnit`, which has numerous range values such as `DAYS`, `HOURS`, `MINUTES`, etc. (actually ranging from `NANOS` for nanoseconds up to `MILLENIA`, `ERAS`, and even `FOREVER`). If you want difference information in a certain unit:

```
jshell> import java.time.temporal.*;

jshell> ChronoUnit.DAYS.between(LocalDate.now(), LocalDate.parse("2033-03-31"));
$1 ==> 3253

jshell> ChronoUnit.DECADES.between(LocalDate.of(1970,01,01),
  LocalDate.now());
$2 ==> 5
```

Unix has completed five decades of active service and is part way into its sixth decade!

See Also

A higher-level way of formatting date/time values is discussed in Recipe 6.2.

6.6 Adding to or Subtracting from a Date

Problem

You need to add or subtract a fixed period to or from a date.

Solution

Create a past or future date by using a method such as `LocalDate.plus(Period.ofDays(N));`.

Discussion

As we've seen, the `Period` class represents a length of time, such as a number of days. `LocalDate` and friends offer a series of `plus()` and `minus()` methods to add or subtract a `Period` or other time-related object, e.g., `plusDays(5)`. `Period` offers factory methods such as `ofDays()`. The following code computes what the date will be 700 days from now:

```
import java.time.LocalDate;
import java.time.Period;

/** DateAdd -- compute the difference between two dates
 * (e.g., today and 700 days from now).
 */
public class DateAdd {
  public static void main(String[] av) {
    /* Today's date */
    LocalDate now =  LocalDate.now();

    LocalDate then1 = now.plusDays(700);
    Period p = Period.ofDays(700);
    LocalDate then2 = now.plus(p);

    System.out.printf("Seven hundred days from %s is %s or %s%n",
      now, then1, then2);
  }
}
```

Running this program reports the current date and time and what the date and time will be 700 days from now:

```
Seven hundred days from 2024-05-03 is 2026-04-03
```

6.7 Calculating Recurring Events

Problem

You need to deal with recurring dates, for example, the third Wednesday of every month.

Solution

Use the `TemporalAdjusters` class.

Discussion

The `TemporalAdjuster` interface and the `TemporalAdjusters` factory class provide most of what you need for recurring events. There are many interesting and powerful adjusters available, shown in Table 6-8, and you can, of course, develop your own.

Table 6-8. New date/time API: `TemporalAdjusters` factory methods

Method signature
`public static TemporalAdjuster firstDayOfMonth();`
`public static TemporalAdjuster lastDayOfMonth();`
`public static TemporalAdjuster firstDayOfNextMonth();`
`public static TemporalAdjuster firstDayOfYear();`
`public static TemporalAdjuster lastDayOfYear();`
`public static TemporalAdjuster firstDayOfNextYear();`
`public static TemporalAdjuster firstInMonth(java.time.DayOfWeek);`
`public static TemporalAdjuster lastInMonth(java.time.DayOfWeek);`
`public static TemporalAdjuster dayOfWeekInMonth(int, java.time.DayOfWeek);`
`public static TemporalAdjuster next(java.time.DayOfWeek);`
`public static TemporalAdjuster nextOrSame(java.time.DayOfWeek);`
`public static TemporalAdjuster previous(java.time.DayOfWeek);`
`public static TemporalAdjuster previousOrSame(java.time.DayOfWeek);`
`public static TemporalAdjuster ofDateAdjuster(` `  java.util.function.UnaryOperator<java.time.LocalDate>);`

The names of most of these tell you directly what they do. The last one will make sense after reading about functional interfaces such as `UnaryOperator` in Chapter 9.

These are used with the `with()` method of a date/time object. For example, the GTABUG group (*https://gtabug.org*) meets on the third Wednesday of every month. I have a `RecurringEventDatePicker` class in the darwinsys-api library; the core

of it started as the method `getMeetingDateInMonth(LocalDate dateContainingMonth)`, which in our case picks the third Wednesday of a given month (given that *`dayOfWeek`* and *`weekOfMonth`* are both set in the constructor). The file *MeetingDates.java* shown in Example 6-6 is a simplified version of this code. This uses the `firstInMonth()` factory method to get a temporal adjuster, then adds the number of weeks to get the Wednesday in the correct week. Its only complexity is that, for the current month, the meeting might be before or after today's date, so we adjust accordingly.

Example 6-6. main/src/main/java/datetime/MeetingDates.java

```
public static LocalDate getNextMeeting(
    int weekOfMonth,
    DayOfWeek dayOfWeek,
    int meetingsAway) {
  LocalDate now = LocalDate.now();
  LocalDate thisMeeting = now.with(
    TemporalAdjusters.dayOfWeekInMonth(weekOfMonth,dayOfWeek));
  // Has the meeting already happened this month?
  if (thisMeeting.isBefore(now)) {
    // start from next month
    meetingsAway++;
  }
  if (meetingsAway > 0) {
    thisMeeting = thisMeeting.plusMonths(meetingsAway).
      with(TemporalAdjusters.dayOfWeekInMonth(weekOfMonth,dayOfWeek));
  }
  return thisMeeting;
}
```

The `main` method that invokes it is shown in Example 6-7.

Example 6-7. main/src/main/java/datetime/MeetingDates.java

```
public static void main(String[] args) {
  DateTimeFormatter dfm = DateTimeFormatter.ofPattern("MMMM dd, yyyy");
  for (int monthAway = 0; monthAway <= 2; monthAway++) {
    LocalDate dt = getNextMeeting(weekOfMonth, dayOfWeek, monthAway);
    System.out.println(dt.format(dfm));
  }
}
```

When visiting this site in Summer 2024, you might have seen something like this:

```
$ java MeetingDates.java
July 17, 2024
August 21, 2024
September 18, 2024
```

6.8 Computing Dates Involving Time Zones

Problem

Imagine a problem like "Your kids are traveling on a transatlantic flight from Toronto to London that takes 5 hours 10 minutes from the actual time of departure from YYZ. Your in-laws need one hour to get to LHR and find parking. What time should you phone them to leave for the airport?"

Solution

The solution needs to take account of time zone differences. It can be solved using the `ZonedDateTime` class and methods such as `plus()` and `minus()` from that class.

Discussion

The basic steps are shown in Example 6-8.

Example 6-8. main/src/main/java/datetime/FlightArrivalTimeCalc.java

```
public class FlightArrivalTimeCalc {

  static Duration driveTime = Duration.ofHours(1);

  public static void main(String[] args) {
    LocalDateTime when = null;
    if (args.length == 0) {
      when = LocalDateTime.now();                                    ❶
    } else {
      String time = args[0];
      LocalTime localTime = LocalTime.parse(time);
      when = LocalDateTime.of(LocalDate.now(), localTime);           ❶
    }
    calculateArrivalTime(when);
  }

  public static ZonedDateTime calculateArrivalTime(LocalDateTime takeOffTime) {
    ZoneId torontoZone = ZoneId.of("America/Toronto"),
        londonZone = ZoneId.of("Europe/London");
    ZonedDateTime takeOffTimeZoned =
      ZonedDateTime.of(takeOffTime, torontoZone);                    ❷
    Duration flightTime =
      Duration.ofHours(5).plus(10, ChronoUnit.MINUTES);              ❸
    ZonedDateTime arrivalTimeUnZoned = takeOffTimeZoned.plus(flightTime); ❹
    ZonedDateTime arrivalTimeZoned =
      arrivalTimeUnZoned.toInstant().atZone(londonZone);             ❺
    ZonedDateTime phoneTimeHere = arrivalTimeUnZoned.minus(driveTime); ❻

    System.out.println("Flight departure time " + takeOffTimeZoned);
```

```
    System.out.println("Flight expected length: " + flightTime);
    System.out.println(
      "Flight arrives there at " + arrivalTimeZoned + " London time.");
    System.out.println("You should phone at " + phoneTimeHere + " Toronto time");
    return arrivalTimeZoned;
  }
}
```

❶ Get the departure time as a `LocalDateTime` (defaulting to `now()` if no arguments passed into `main()`, on the assumption that we run the app when the flight takes off).

❷ Convert departure time to `ZonedDateTime`.

❸ Convert flight time to a `Duration`.

❹ Get the arrival time by adding the departure time to the flight duration.

❺ Convert the arrival time to London time with `atZone()`.

❻ Since the family takes an hour to get to the airport, subtract that from the arrival time. This yields the time when you should phone them.

6.9 Interfacing with Legacy Date and Calendar Classes

Problem

You need to deal with the old `Date` and `Calendar` classes.

Solution

Assuming you have code using the original `java.util.Date` and `java.util.Calendar`, you can convert values as needed using conversion methods.

Discussion

All the classes and interfaces in the new API were chosen to avoid conflicting with the traditional API. It is thus possible, and will be common for a while, to have imports from both packages in the same code.

To keep the new API clean, most of the necessary conversion routines were *added to the old API*. Table 6-9 summarizes these conversion routines; note that the methods are static if they are shown being invoked with a capitalized class name, otherwise they are instance methods.

Table 6-9. Legacy date/time interchange

Legacy class	Convert to legacy	Convert to modern
`java.util.Date`	`date.from(Instant)`	`Date.toInstant()`
`java.util.Calendar`	`calendar.toInstant()`	-
`java.util.GregorianCalendar`	`GregorianCalendar.from(ZonedDateTime)`	`calendar.toZonedDateTime()`
`java.util.TimeZone`	-	`timeZone.toZoneId()`
`java.text.Format`	`dateTimeFormatter.toFormat()`	n/a

Example 6-9 shows some of these APIs in action.

Example 6-9. main/src/main/java/datetime/LegacyDates.java

```
public class LegacyDates {
  public static void main(String[] args) {

    // There and back again, via Date
    Date legacyDate = new Date();
    System.out.println(legacyDate);

    LocalDateTime newDate =
      LocalDateTime.ofInstant(legacyDate.toInstant(),
      ZoneId.systemDefault());
    System.out.println(newDate);

    Date backAgain =
      Date.from(newDate.atZone(ZoneId.systemDefault()).toInstant());
    System.out.println("Converted back as " + backAgain);

    // And via Calendar
    Calendar c = Calendar.getInstance();
    System.out.println(c);
    LocalDateTime newCal =
      LocalDateTime.ofInstant(c.toInstant(),
      ZoneId.systemDefault());
    System.out.println(newCal);
  }
}
```

You do not have to use these legacy converters; you are free to write your own, though doing so is usually a fool's errand. The file *LegacyDatesDIY.java* in the *javasrc* repository explores this option in the unlikely event you wish to pursue it.

Given the amount of code written before Java 8, it is likely that the legacy `Date` and `Calendar` will be around until the end of Java time.

The new date/time API has many capabilities that we have not explored. Almost enough for a small book on the subject, in fact. Meanwhile, you can study the API details at Oracle.

CHAPTER 7

Structuring Data with Java

7.0 Introduction

Almost every application beyond "Hello, World" needs to keep track of some structured data. A simple numeric problem might work with three or four numbers only, but most applications have groups of similar data items. A GUI-based application may need to keep track of a number of dialog windows. A personal information manager, or PIM, needs to keep track of a number of, well, persons. An operating system needs to keep track of who is allowed to log in, who is currently logged in, and what those users are doing. A library needs to keep track of who has books checked out and when they're due. A network server may need to keep track of its active clients. A pattern emerges here, and it revolves around variations of what has traditionally been called *data structuring*.

There are data structures in the memory of a running program; there is structure in the data in a file on disk, and there is structure in the information stored in a database. In this chapter, we concentrate on the first aspect: in-memory data. We'll cover the second aspect in Chapter 10; the third is out of scope for this book.

If you had to think about in-memory data, you might want to compare it to a collection of index cards in a filing box or to a treasure hunt where each clue leads to the next. Or you might think of it like my desk—apparently a scattered mess, but actually (I claim) a very powerful collection filled with meaningful information. Each of these is a good analogy for a type of data structuring that Java provides. An *array* is a fixed-length linear collection of data items, like the card filing box: it can only hold so much, then it overflows. The treasure hunt is like a data structure called a *linked list*. The first release of Java had no standard linked list class, but you could write your own traditional data structure classes (and still can; there's a demo implementation of a simplified linked list [not production ready] to show how this kind of list works, in

main/src/main/java/structure/LinkList.java). A third category is *associative* or *map* types, also known as "key/value" collections, where each key maps to a distinct value. The variety of collections is represented by Java's `Collections` classes. A document entitled "Collections Framework Overview" exists in the Java documentation; unfortunately these tutorials have not been updated since Java 8, so they should be taken with a grain of salt. They do provide a detailed discussion of the Collections Framework as it was in those days. The framework aspects of Java collections are summarized in Recipe 7.4.

Beware of typographic issues. The word `Arrays` (in constant width font) refers to the utility class `java.util.Arrays`; but in the normal typeface, the word "arrays" is simply the plural of "array" (and will be found capitalized at the beginning of a sentence). Similarly, both `Collections` and `Collection` are types in the framework; the former is a utility class and the latter is an interface implemented by the nonassociative `Collection` classes. Also, note that `HashMap` and `HashSet` follow the rule of having a midcapital at each word boundary, whereas the older `Hashtable` does not (the *t* is not capitalized).

The java.util package has become something of a catch-all over the years. Besides the legacy date/time API covered in Recipe 6.9, several other classes from java.util are not covered in this chapter. All the classes whose names begin with `Abstract` are, in fact, abstract, and we'll discuss their nonabstract subclasses. The `StringTokenizer` class is covered in Recipe 3.1. `BitSet` is used less frequently than some of the classes discussed here and is simple enough to learn on your own. `BitSet` stores the bits very compactly in memory, but because it predates the Collection API and wasn't retrofitted, it doesn't implement the standard collection interfaces. Also not covered here are `EnumSet` and `EnumMap`, specialized for efficient storage/retrieval of enums. These are newer than `BitSet` and *do* implement the modern collection interfaces.

We start our discussion of data structuring techniques with one of the oldest structures, the array. We'll discuss the overall structure of java.util's Collections Framework, then go through a variety of structuring techniques using classes from java.util.

7.1 Using Arrays for Data Structuring

Problem

You need to keep track of a fixed amount of information and retrieve it (usually) sequentially.

Solution

Use an array.

Discussion

Arrays can be used to hold any linear collection of data. You can make an array of any primitive type or any object type. For *arrays of primitive types*, such as `int`s and `boolean`s, the data is stored in the array. For *arrays of objects*, a reference is stored in the array, so the normal rules of reference variables and casting apply. The items in an array of primitives must all be of the same type, while those in an object array should generally all be of the same type (allowing for subtypes). Note in particular that if the array is declared as `Object[]`, object references of any type can be stored in it without casting, although a valid cast is required to take an `Object` reference out and use it as its original type. I'll say a bit more on two-dimensional arrays in Recipe 7.15; otherwise, you should treat Example 7-1 as a review example.

Example 7-1. main/src/main/java/lang/Array1.java

```
public class Array1  {
  @SuppressWarnings("unused")
  public static void main(String[] argv) {
    int[] monthLen1;      // declare a reference
    monthLen1 = new int[12];    // construct it
    int[] monthLen2 = new int[12];  // short form
    // even shorter is this initializer form:
    int[] monthLen3 = {
        31, 28, 31, 30,
        31, 30, 31, 31,
        30, 31, 30, 31,
    };

    final int MAX = 10;
    LocalDate[] days = new LocalDate[MAX];
    for (int i=0; i<MAX; i++) {
      days[i] = LocalDate.of(2022, 02, i + 1);
    }

    // Two-Dimensional Arrays
    // Want a 10-by-24 array
    int[][] me = new int[10][];
    for (int i=0; i<10; i++)
      me[i] = new int[24];

    // Remember that an array has a ".length" attribute
    System.out.println(me.length);
    System.out.println(me[0].length);

  }
}
```

Arrays in Java work well. The type checking provides reasonable integrity, and array bounds are always checked by the runtime system, further contributing to reliability. Arrays are objects, but with no additional methods. An array has a (read-only) `length` attribute, distinct for each individual array; the length is not part of the type.

See Also

The only problem with arrays is: what if the array fills up and you still have data coming in? See Recipe 7.2.

7.2 Resizing an Array

Problem

The array filled up, and you got an `ArrayIndexOutOfBoundsException`.

Solution

Make the array bigger by reallocating and copying the values.

Discussion

One approach is to allocate the array at a reasonable size to begin with; but if you find yourself with more data than will fit, reallocate a new, bigger array and copy the elements into it.[1] Example 7-2 contains code that does so.

Example 7-2. main/src/main/java/structure/Array2.java

```
public class Array2  {
    public final static int INITIAL = 10,    ❶
        GROW_FACTOR = 2;                      ❷

    public static void main(String[] argv) {
        int nDates = 0;
        LocalDateTime[] dates = new LocalDateTime[INITIAL];
        StructureDemo source = new StructureDemo(21);
        LocalDateTime c;
        while ((c=source.getDate()) != null) {

            // Suboptimal: give up
            // if (nDates >= dates.length) {
            //     throw new RuntimeException(
            //         "Too Many Dates! Simplify your life!!");
```

1 You could copy it yourself using a `for` loop if you wish, but `System.arrayCopy()` is likely to be faster because it's implemented in native code.

```
            // }

            // Better: reallocate, making data structure dynamic
            if (nDates >= dates.length) {
                LocalDateTime[] tmp =
                    new LocalDateTime[dates.length * GROW_FACTOR];
                System.arraycopy(dates, 0, tmp, 0, dates.length);
                dates = tmp;    // copies the array reference
                // old array will be garbage collected soon...
            }
            dates[nDates++] = c;
        }
        System.out.println("Final array size = " + dates.length);
    }
}
```

❶ A good guess is necessary; know your data!

❷ The growth factor is arbitrary; 2 is a good value here but will continue to double exponentially. You might want to use a factor like 1.5, which would mean more allocations at the low end but less explosive growth. You need to manage this somehow!

This technique works reasonably well for simple or relatively small linear collections of data. For data with a more variable structure, you probably want to use a more dynamic approach, as in Recipe 7.5.

7.3 Simplifying Array Handling with the Arrays Class

Problem

You need to format an array, or to compare two arrays.

Solution

Somebody suggested that you use a `for` loop. In many cases, you should not do this, but use the `Arrays` class instead:

- `Arrays.toString()` to print the values as a list
- `Arrays.equals()` for simple equality
- `Arrays.compare()` for lexicographical ("high-equals-low") comparison

Discussion

The `Arrays` class has a number of utility methods for dealing with array data. It's odd that Java language arrays, although they are objects, do not have a useful `toString()`

method; that is left to the static `Arrays.toString()` method, of which there are nine overloads. There are similarly nine overloads of the static `equals()` method for comparing arrays for equality. There are 28 overloads of `compare`-type methods just for comparing arrays lexicographically!

The `static toString()` method has nine overloads, for arrays of any primitive type or arrays of type `Object`. The elements of the array are printed as a list, in square brackets, separated by a comma and a space. Example 7-3 demonstrates this.

Example 7-3. main/src/main/java/structure/ArraysToString.java

```
record Cat(String name){
  public String toString() { return "Meower(" + name + ")";}
}
Cat[] pets = new Cat[]{new Cat("Tom"), new Cat("Slinky")};
System.out.println("Pets are " + Arrays.toString(pets));
```

If you're unfamiliar with the use of `record` instead of `class`, refer to Recipe 8.4.

When run, Example 7-3 prints:

```
Pets are [Meower(Tom), Meower(Slinky)]
```

The static `equals()` method compares two arrays of the same type:

```
boolean Arrays.equals(a1, a2);
```

Here `a1` and `a2` are arrays of any primitive type, or arrays of (compatible) `Object` types.

The `compare()` method has overloads for all common types:

```
public static int compare(X[], X[]);
public static int compare(X[] a, int aFrom, int aTo, X[] b, int bFrom, int bTo);
```

In this example, `X` is any of the eight primitive types, or an `Object` type. These all return, quoting the JDK documentation, "0 if the first and second array are equal and contain the same elements in the same order; a value less than 0 if the first array is lexicographically less than the second array; and a value greater than 0 if the first array is lexicographically greater than the second array."

For the seven numeric types there are also `compareUnsigned()` methods:

```
public static int compareUnsigned(X[], X[]);
public static int compareUnsigned(X[] a, int aFrom, int aTo, X[], int bFrom,
int bTo);
```

Finally, there are methods for comparing arrays of `Object` types:

```
public static <T extends java.lang.Comparable<? super T>> int compare(T[], T[]);
public static <T extends java.lang.Comparable<? super T>> int compare(T[],
  int, int, T[], int, int);
public static <T> int compare(T[], T[], java.util.Comparator<? super T>);
public static <T> int compare(T[], int, int, T[], int, int,
  java.util.Comparator<? super T>);
```

Here the `T` is a *generic type*; see Recipe 7.5 if you're not familiar with the concept.

The `equals()`, `compare()`, and `toString()` methods are illustrated in Example 7-4.

Example 7-4. main/src/main/java/structure/ArraysEqualsCompareToString.java

```
public class ArraysEqualsCompareToString {

    void main() {
        System.out.println("Permute int value");
        int[] ia = { 1, 2, 3, 4, 5},
              ib = { 1, 2, 3, 4, 5};
        System.out.println(
            "ia = " + Arrays.toString(ia) +
            "ib = " + Arrays.toString(ib));
        System.out.println("By Arrays.equals(): " + Arrays.equals(ia, ib));
        System.out.println("compare(ia,ib) returns " + Arrays.compare(ia, ib));
        ia[4] = 6;
        System.out.println(
            "ia = " + Arrays.toString(ia) + ", " +
            "ib = " + Arrays.toString(ib));
        System.out.println("By Arrays.equals(): " + Arrays.equals(ia, ib));
        System.out.println("compare(ia,ib) returns " + Arrays.compare(ia, ib));
```

To save a tree, this shows one segment of the code; it is repeated with an array of a different type, and another with the contents of one element shuffled. The output of this segment is as follows:

```
Permute int value
ia = [1, 2, 3, 4, 5], ib = [1, 2, 3, 4, 5]
By Arrays.equals(): true
compare(ia,ib) returns 0
ia = [1, 2, 3, 4, 6], ib = [1, 2, 3, 4, 5]
By Arrays.equals(): false
compare(ia,ib) returns 1
```

7.4 The Collections Framework

Problem

You're having trouble keeping track of all these lists, sets, and iterators.

Solution

There's a pattern to it. See Table 7-1 and Figure 7-1.

Discussion

`List`, `Set`, `Map`, and `Queue` are the four fundamental data structures of the Collections Framework. `List` and `Set` are both sequences, with the difference that `List` preserves order and allows duplicate entries, whereas `Set`, true to the mathematical concept behind it, does neither, although there is one implementation of `Set` that preserves order. `Map` is a key/value store, also known as a hash, a dictionary, or an associative store. `Queues` are, as the name suggests, structures that you can push into at one end and pull out from the other.

Table 7-1 shows some of the important collection-based classes from package java.util. It is far from complete, due to space limitations. Figure 7-1 shows the relationships among some of the more important types in the framework. Rectangles are interfaces; ovals, classes. Solid lines are inheritance; dashed lines represent implementations of interfaces. `Queue` and its subtypes are treated in Chapter 11.

Table 7-1. Java collections

Interfaces	Implementations			
	Resizable array	Hashed table	Linked list	Balanced tree
`List`	`ArrayList, Vector`		`LinkedList`	
`Set`		`HashSet`		`TreeSet`
`Map`		`HashMap, Hashtable`		`TreeMap`
`Queue`	`Deques, BlockingQueues, etc.`			

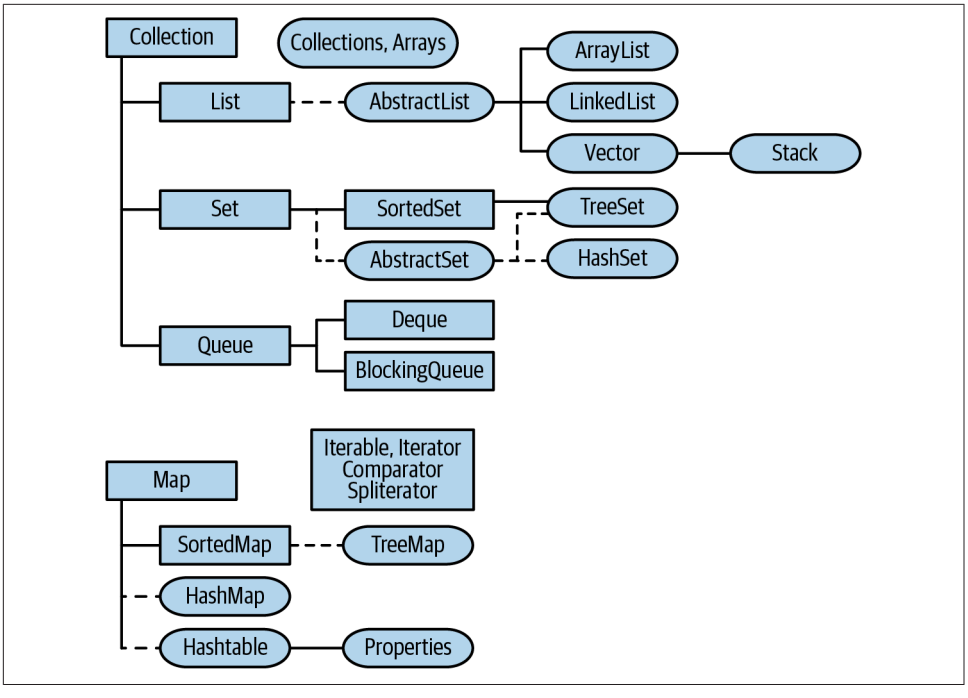


Figure 7-1. The Collections Framework

See Also

The Javadoc documentation on `Collections`, `Arrays`, `List`, `Set`, and the classes that implement them provides more details than there's room for here. Table 7-1 may further help you to absorb the regularity of the Collections Framework.

7.5 Lists: Like an Array, but More Dynamic

Problem

You don't want to worry about storage reallocation (often because you don't know how big the incoming dataset is going to be); you want a standard class to handle it for you. You want to store your data in any of the `Collections` classes defined in this chapter with type safety and without having to write downcasts when retrieving data from the collection.

Solution

Use a `List` implementation or one of the other `Collections` classes, along with Java's generic types mechanism, declaring the `Collection` with a *type parameter* identifying

the type of your data. The type parameter name appears in angle brackets after the declaration, and optionally on the instantiation.

Discussion

The first of the `Collections` interface we will discuss, `List`, encapsulates the functionality of an array but allows it to expand automatically. You can just keep on adding things to it, and each addition behaves the same. Two main implementations are the `ArrayList` and the `LinkedList`.

Because the `List` implementations are classes and not part of the syntax of Java, you can't use Java's array syntax; you must use methods to access the `List`'s data. The interface has methods to add objects, retrieve objects, find objects, and tell you how big the `List` is. The `ArrayList` class has methods to say and control how big it can become without having to reallocate.

21 Java 21 adds some convenience methods: `addFirst()`, `addLast(E)`, `getFirst()`, `getLast()`, `removeFirst()`, and `removeLast()`.

Like the other `Collections` classes in java.util, `ArrayList`'s storing and retrieval methods were originally defined to have parameters and return values of `java.lang.Object`. Because `Object` is the ancestor of every defined type, you can store objects of any type in a `List` (or any collection) and cast it when retrieving it. If you need to store a small number of primitives (like `int` or `float`) in a collection, use the appropriate wrapper class (usually by auto-boxing; see the introduction to Chapter 5). To store `boolean`s, either store them directly in the specialized `java.util.BitSet` (which is neither a `List` nor a `Set` implementation; see the online documentation) or store them in a `List` using the `Boolean` wrapper class.

Because `Object` is usually too general for accurate work, Java provides the *generic types* mechanism. You declare an `ArrayList` (or other collection) with a type parameter in angle brackets, and the parameters and returns are treated as being of that type by the compiler. This ensures that objects of the wrong type don't make it into your collections, and avoids the need to write casts when retrieving objects. For example, this is how you declare an `ArrayList` for holding `String` object references:

```
List<String> myList = new ArrayList<>();
```

It is a good practice to *declare* a field as the interface type `List`, even though you are *defining* it (constructing it) as an `ArrayList`. This makes it easier to change from one `List` implementation to another, and it avoids accidentally depending on an implementation-specific method not in the `List` interface (which would also make it harder to change the implementation).

The <> in the definition part is a vestige of legacy Java versions, in which you had to repeat the type definition, so you'd write `new ArrayList<String>()` in that example. Nowadays just use <> (as in the example) to indicate that you want the type copied from the declaration. The <> is called the *diamond operator*.

As of Java 13, for local variables (inside a method) you can simplify by using the new `var` keyword; here, put the type parameter on the definition:

```
var myList = new ArrayList<String>();
```

Here the variable will be of type `ArrayList`, so it doesn't have the `List`-only semantics, but there is still only one place to change if you want to switch to, say, a `LinkedList`.

Table 7-2 shows some of the most important methods of the `List` interface, which is implemented by `ArrayList`, `LinkedList`, and `Vector`. This means that the same methods can be used with the older `Vector` class and any other `List`-implementing class. You'd just have to change the name used in the constructor call.

Table 7-2. Common `List<T>` methods

Method signature	Usage
`add(T o)`	Add the given element at the end
`add(int i, T o)`	Insert the given element at the specified position
`clear()`	Remove *all* element references from the `Collection`
`contains(T o)`	True if the `List` contains the given object
`forEach(lambda)`	Perform the lambda (see Chapter 9) for each element
`get(int i)`	Return the object reference at the specified position
`indexOf(T o)`	Return the index where the given object is found, or –1
`of(T t, …)`	Create a list from multiple objects
`remove(T o), remove(int i)`	Remove an object by reference or by position
`toArray()`	Return an array containing the objects in the `Collection`

Example 7-5 stores data in an `ArrayList` and retrieves it for processing.

Example 7-5. main/src/main/java/structure/ArrayListDemo.java

```
public class ArrayListDemo {
  public static void main(String[] argv) {
    List<LocalDate> editions = new ArrayList<>();

    // Add lots of elements to the ArrayList...
    editions.add(LocalDate.of(2001, 06, 01));
    editions.add(LocalDate.of(2004, 06, 01));
    editions.add(LocalDate.of(2014, 06, 20));
```

```
    // Use old-style 'for' loop to get index number.
    System.out.println("Retrieving by index:");
    for (int i = 0; i<editions.size(); i++) {
      System.out.printf("Edition %d was %s\n", i + 1, editions.get(i));
    }
    // Use normal 'for' loop for simpler access.
    System.out.println("Retrieving by Iterable:");
    for (LocalDate dt : editions) {
      System.out.println("Edition " + dt);
    }

  }
}
```

The older `Vector` class predates the Collections Framework and offers additional methods with different names: `Vector` provides `addElement()` and `elementAt()`. You might still run across these in legacy code, but you should use the `List` interface methods `add()` and `get()` instead.

Another difference is that the methods of `Vector` are synchronized, so they can be accessed safely from multiple threads (see Recipe 11.4). This does mean more overhead, though, so for single-threaded access it is faster to use an `ArrayList`.

There are various conversion methods. Table 7-2 mentions `toArray()`, which will expose the contents of a `List` as an array. The `List` interface in Java 9+ features a static `of()` method, which converts in the other direction, from an array to a `List`. In conjunction with the variable arguments feature of modern Java, you can create and populate a list in one call to `List.of()`, like this:

```
List<String> firstNames = List.of("Robin", "Jaime", "Joey");
```

In legacy code that you will find in older apps and in web searches, `Arrays.asList()` provided this functionality, so you will come across code like this:

```
// Older way; no longer best
List<String> lastNames = Arrays.asList("Smith", "Jones", "MacKenzie");
// or even
List<String> lastNames =
    Arrays.asList(new String[]{"Smith", "Jones", "MacKenzie"});
```

Java does indeed get less verbose as time goes by!

Due to Java's insistence on backward compatibility—the ability to run old code on new Java versions—you can still instantiate `Collections` classes without providing a type parameter. If you do, you will get a compiler warning, and the class will behave as in the very early days; that is, the objects returned from any object-returning

method will be treated as of type `java.lang.Object`, and must be downcast before you can call any class-specific methods or use them in any application-specific method calls.

A *linked list* is commonly used when you have an unpredictably large number of data items, want to allocate just the right amount of storage, and expect to access them primarily in the order that you created them. This structure also allows for efficient insertion and removal, since the items after the change do not have to be moved, only a couple of references need to be adjusted. Each element contains both the actual data item and a reference to the "next" item, with null for the last reference at the end of the list. Figure 7-2 shows the normal arrangement.

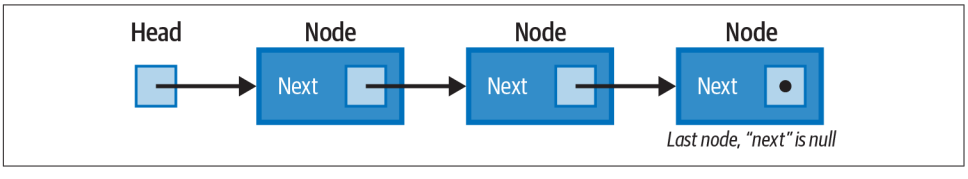


Figure 7-2. Linked list structure

The Collections API provides a `LinkedList` class; Example 7-6 is a simple program that uses it.

Example 7-6. main/src/main/java/structure/LinkedListDemo.java

```
public class LinkedListDemo {
  public static void main(String[] argv) {
    System.out.println("Here is a demo of Java's LinkedList class");
    LinkedList<String> l = new LinkedList<>();
    l.add(new Object().toString());
    l.add("Hello");
    l.add("end of the list");

    System.out.println("Here is a list of all the elements");
    l.forEach(o ->
      System.out.println("Next element: " + o));

    if (l.indexOf("Hello") != 1)
      System.err.println("Lookup does not work");
    else
      System.err.println("Lookup works");

    // Now, for added fun, let's walk the linked list backward.
    ListIterator<String> li = l.listIterator();
    // (have to get to the end first)
    while (li.hasNext())
      li.next();
    while (li.hasPrevious()) {
      System.out.println("Backwards we find: " + li.previous());
```

```
    }
  }
}
```

The `ListIterator` used here is a subinterface of `Iterator`, discussed in Recipe 7.7.

As a further example of type parameterization, consider the `Map` interface mentioned in Recipe 7.4. A `Map` requires a key and a value in its `put()` method. A `Map`, therefore, has two parameterized types. To set up a `Map` whose keys are `Person` objects and whose values are `Address` objects (assuming these two classes exist in your application), you could define it like this if it's a field:

```
Map<Person, Address> addressMap = new HashMap<>();
```

or, if it's a local variable:

```
var addressMap = new HashMap<Person, Address>();
```

This `Map` expects a `Person` as its key and an `Address` as its value in the `put()` method. The `get()` method takes one argument of type `Person` and returns an `Address` object. The `keySet()` method returns `Set<Person>` (i.e., a `Set` specialized for `Person` objects). There are also convenience routines to create a `Map` from existing objects. The most useful is several overloads of `of`, such as `Map.of(key,value,key,value…)`, similar to `List.of()` (but limited to 10 pairs), and so on.

See Also

Although the generics avoid the need to write downcasts, the casts still occur at runtime; they are just provided by the compiler. The compiler techniques used in compiling these new constructs in a backward-compatible way include *erasure* and *bridging*, topics discussed in *Java Generics and Collections* by Maurice Naftalin and Philip Wadler (O'Reilly).

7.6 Using Generic Types in Your Own Class: Stack Demo

Problem

You wish to define your own container classes using the generic type mechanism to avoid needless casting.

Solution

Define a class using < *TypeName* > where the container type is declared and *TypeName* where it is used.

Discussion

I'll demonstrate the notion of creating a generic type using a reimplementation of the `Stack` class. A criticism of the standard `Stack` class in java.util is that it extends `Vector`, such that you can remove elements from the middle of the stack, which is not "normal" for a `Stack`.

A *stack*, or more precisely a *push-down stack*, is a common data structuring operation and is often used to reverse the order of objects. The name is an analogy to a stack of papers on your desk; you add things on top and remove them from the top. A common software example is an *undo* stack in an editor, IDE, or word processor: it lets you undo edits in last-in, first-out (LIFO) order. The basic operations of a stack (in any programming language) are:

`push()`
: Add to stack

`pop()`
: Remove from stack

`peek()`
: Examine top element without removing

I built several versions of my own stack, culminating in the code in Example 7-7. Since the original `Stack` defines no interface of its own, I made an interface for my implementations, called `SimpleStack` to differentiate from the original. My stack has a method not in the original, `hasRoom()`. Unlike the full-blown `java.util.Stack`, `MyStack` does not expand beyond its original size, so we need a way to see if it is full without throwing an exception.

My interface and class are parameterized to take a "generic type" whose local name is `T`. This type `T` will be the type of the argument of the `push()` method, the return type of the `pop()` method, and so on. Because of this specific return type, the return value from `pop()` does not need to be downcasted. The `Container` classes in the Collections Framework (`java.util`) are parameterized similarly.

Example 7-7. main/src/main/java/structure/MyStack.java

```
public class MyStack<T> implements SimpleStack<T> {

  private int depth = 0;
  public static final int DEFAULT_INITIAL = 10;
  private T[] stack;

  public MyStack() {
    this(DEFAULT_INITIAL);
  }
```

```
  public MyStack(int howBig) {
    if (howBig <= 0) {
      throw new IllegalArgumentException(
      howBig + " must be positive, but was " + howBig);
    }
    stack = (T[])new Object[howBig];
  }

  @Override
  public boolean empty() {
    return depth == 0;
  }

  /** push - add an element onto the stack */
  @Override
  public void push(T obj) {
    // Could check capacity and expand
    stack[depth++] = obj;
  }

  /* pop - return and remove the top element */
  @Override
  public T pop() {
    --depth;
    T tmp = stack[depth];
    stack[depth] = null;
    return tmp;
  }

  /** peek - return the top element but don't remove it */
  @Override
  public T peek() {
    if (depth == 0) {
      return null;
    }
    return stack[depth-1];
  }

  public boolean hasNext() {
    return depth > 0;
  }

  public boolean hasRoom() {
    return depth < stack.length;
  }

  public int getStackDepth() {
    return depth;
  }
}
```

The association of a particular type is done at the time the class is instantiated. For example, to instantiate a `MyStack` specialized for holding `BankAccount` objects, you would need to code only the following:

```
MyStack<BankAccount> theAccounts = new MyStack<>();
```

If you don't provide a type parameter `T`, this collection, like the ones in `java.util`, will behave as they did in the days before generic collections—accepting input arguments of any type, returning `java.lang.Object` from getter methods, and requiring downcasting—as their default, backward-compatible behavior. Example 7-8 shows a program that creates two instances of `MyStack`, one specialized for `Strings` and one left general. The general one, called `ms2`, is loaded up with the same two `String` objects as `ms1` but also includes a `Date` object. The printing code is now broken, because it will throw a `ClassCastException`: a `Date` is not a `String`. I handle this case specially for pedantic purposes: it is illustrative of the reason generic types were added to Java, and of the kinds of errors you can get into when using nonparameterized container classes.

Example 7-8. main/src/main/java/structure/MyStackDemo.java

```
public class MyStackDemo {

  @SuppressWarnings({"rawtypes","unchecked"})
  public static void main(String[] args) {
    MyStack<String> ms1 = new MyStack<>();
    ms1.push("billg");
    ms1.push("scottm");

    while (ms1.hasNext()) {
      String name = ms1.pop();
      System.out.println(name);
    }

    // Old way of using Collections: not type safe.
    // DO NOT GENERICIZE THIS
    MyStack ms2 = new MyStack();
    ms2.push("billg");               // EXPECT WARNING
    ms2.push("scottm");              // EXPECT WARNING
    ms2.push(new java.util.Date());  // EXPECT WARNING

    // Show that it is broken
    try {
      String bad = (String)ms2.pop();
      System.err.println("Didn't get expected exception, popped " + bad);
    } catch (ClassCastException ex) {
      System.out.println("Did get expected exception.");
    }

    // Removed the brokenness, print rest of it.
```

```
    while (ms2.hasNext()) {
      String name = (String)ms2.pop();
      System.out.println(name);
    }
  }
}
```

Because of this potential for error, the compiler warns that you have unchecked raw types. Like the deprecation warnings discussed in Recipe 2.7, by default, these warnings are not printed in detail by the `javac` compiler (they will appear in most IDEs). You ask for them with the rather lengthy option `-Xlint:unchecked`, but they appear automatically in JShell:

```
jshell> var list = new ArrayList();
list ==> []

jshell> list.add("Hello");
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  list.add("Hello");
|  ^---------------^
$2 ==> true
jshell>
```

7.7 How Shall I Iterate Thee? Let Me Enumerate the Ways

Problem

You need to iterate over some structured data.

Solution

Java provides many ways to iterate over collections of data. Here they are, in newest-first order:

- `Stream.forEach()` method 8
- `IntStream.rangeClosed(start, end).forEach(Consumer c)` 8
- `Iterable.forEach()` method 8
- for-each loop 5
- `java.util.Iterator`
- `java.util.Enumeration`
- Three-part `for` loop (borrowed from 1970s C language)
- `while` loop (ditto)

Pick one and use it. Or learn them all and save!

Discussion

A few words on each of the iteration methods are given here. Note that the first few are the most common.

Stream.forEach method 8

The `Stream` mechanism introduced as part of Java's functional programming provides one of the two most recent ways of iterating, `Stream.forEach()`, and is discussed in Recipe 9.3. For now, here's a quick example, using the `BufferedReader` method `lines()` that returns a `Stream`:

```
$ jshell
jshell> import java.io.*;
jshell> BufferedReader is =
  new BufferedReader(new FileReader("/home/ian/.profile"));
is ==> java.io.BufferedReader@58651fd0
jshell> is.lines().forEach(System.out::println)
... prints the lines of the file ...
```

IntStream.rangeClosed().forEach() 8

This method allows you to iterate over a range of integer numbers. The `rangeClosed()` method is described in Recipe 5.7 and the `Consumer` interface is covered in Chapter 9.

Iterable.forEach method 8

The other recent iteration technique is the `Iterable.forEach()` method, added in Java 8. This method can be called on any `Iterable` (unfortunately, the array class does not yet implement `Iterable`) and takes one argument implementing the *functional interface* `java.util.function.Consumer`. Functional interfaces are discussed in Chapter 9, but here is one example:

```
public class IterableForEach {

    public static void main(String[] args) {
        Collection<String> c =                    ❶
                List.of("One", "Two", "Three");   ❷
        c.forEach(s -> System.out.println(s));    ❸
    }
}
```

❶ Declare a `Collection` (a `Collection` is an `Iterable`).

❷ Populate it with `List.of()` with a sequence of objects (see Recipe 7.5 for how this arbitrary argument list becomes a `List`).

❸ Invoke the collection's `forEach()` method, passing a lambda expression (see Chapter 9 for a discussion of how `s→System.out.println(s)` gets mapped to a `Consumer` interface implementation without the need to import this interface).

This style of iteration—sometimes called *internal iteration*—inverts the control from the traditional `for` loop; the collection is in charge of when and how the iteration works.

Both `Stream.forEach` and `Iterable.forEach()` take one argument, of type `java.util.function.Consumer`, so they work largely the same way, at least syntactically. This is intentional.

Java for-each loop

This is the for-each loop syntax:

```
for (Type var : Iterable<Type>) {
        // do something with "var"
}
```

The for-each loop is probably the most common style of loop in modern Java code. The `Iterable` can be an array or anything that implements `Iterable` (the `Collection` implementations included).

This style is used throughout the book. In addition, many third-party frameworks/libraries provide their own types that implement `Iterable` for use with the `for` loop.

Three-part for loop

This is the traditional `for` loop invented by Dennis Ritchie in the early 1970s for the C language:

```
for (init; test; change) {
        // do something
}
```

Its most common form is with an `int` "index variable" or "loop variable":

```
MyDataType[] data = ...
for (int i = 0; i < data.length; i++)
        MyDataType d = data[i];
        // do something with 'd'
}
```

while loop

A `while` loop executes its loop body as long as (while) the test condition is true. It's commonly used in conjunction with an `Enumeration` or `Iterator`, like this:

```
Iterator<MyData> iterator = ...
while (iterator.hasNext()) {
        MyData md = iterator.next();
        //
}
```

java.util.Iterator

The older `Iterator` interface has three methods:

```
public interface java.util.Iterator<E> {
  public abstract boolean hasNext();
  public abstract E next();
  public default void remove();
}
```

It was once common to write code like this, which you'll still find occasionally in older code:

```
Iterator it = ...; // legacy code; might not even have type parameter
while (it.hasNext()) {
        (MyDataType) c = it.next();
        // Do something with c
}
```

The `remove()` method in the `Iterator` throws an `UnsupportedOperationException` if called on a read-only collection. In conjunction with `Streams` and default methods, there is now a fourth method:

```
public default void forEachRemaining(java.util.function.Consumer<? super E>);
```

Enumeration

An `Enumeration` is like an `Iterator` (shown earlier), but it lacks the `remove()` method, and the control methods have longer names—for example, `hasMoreElements()` and `nextElement()`. For new code, there is little to recommend using `Enumeration`.

7.8 Avoiding Duplicate Values with a Set

Problem

You want a structure that will avoid storing duplicates.

Solution

Use a `Set` implementation instead of a `List` (e.g., `Set<String> myNames = new HashSet<>()`).

Discussion

The `Set` interface is similar to the `List` interface,[2] with methods like `add()`, `remove()`, `contains()`, `size()`, and `isEmpty()`. The difference is that a `Set` doesn't preserve order; instead, it enforces uniqueness—if you add the same item (as considered by its `equals()` method twice or more, it will only be present once in the set. For this reason, the index-based methods such as `add(int, Object)` and `get(int)` are missing from the `Set` interface: you might know that you've added seven objects but only five of those were unique, so calling `get()` to retrieve the sixth one would throw an `ArrayIndexOutOfBoundsException`! It's better not to think of a `Set` as being indexed, because it isn't. Instead, the `contains()` method can be used to determine whether a given object is or is not currently in the `Set`.

As the Java `Set` document states: "Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element." Basically, just store immutable objects in sets.

Example 7-9 shows a duplicate entry being made to a `Set`, which will contain only one copy of the string `"One"`.

Example 7-9. main/src/main/java/structure/SetDemo.java

```
Set<String> hashSet = new HashSet<>();
hashSet.add("One");
hashSet.add("Two");
hashSet.add("One"); // DUPLICATE
hashSet.add("Three");
System.out.println(hashSet);
```

Not surprisingly, only the three distinct values are printed:

```
[One, Two, Three]
```

2 Both `List` and `Set` are subtypes of `Collection`.

The results are not guaranteed to come out in any particular order. But, if you change the order of the add calls, the list does come out in the same order, which is defined as the hashing order. Items are maintained in the order of their `hashCode()` values; the dedicated reader can explore this via the online version of the *SetDemo.java* file.

If you need a `Set` sorted, there is a `SortedSet` interface, of which the most common implementation is `TreeSet`; see a `TreeSet` example in "TreeSet Example". There are constructor overloads that accept a `Comparator`; the default is to use the class's natural order.

As with `Lists`, the `Set` interface offers the `of` method as of Java 9:

```
Set<Double> nums = Set.of(Math.PI, 22D/7, Math.E);
Set<String> firstNames = Set.of("Robin", "Jaime", "Joey");
```

There are many other methods in the `Set` interface. Although there are no methods explicitly named for set union and intersection operations, these are easily implemented using the `addAll()` and `retainAll()` methods in the `Set` interface. An example is shown in *main/src/main/java/structure/SetArithmetic.java*, of both creating temporary `Sets` to hold the result, and using Streams `Collectors` (see Chapter 9) with no temporary objects.

7.9 Mapping with Hashtable and HashMap

Problem

You need a one-way mapping from one data item to another.

Solution

Use a `Map` implementation, such as a `HashMap`.

Discussion

`Maps` provide a one-way mapping from one set of object references to another. They are completely general purpose. I've used them to map from Swing push buttons to the URL that is to be opened when the button is pushed, to map names to addresses, and to implement a simple in-memory cache in a web server. You can map from anything to anything. You'd usually want to provide type parameters for both the keys and the values. Example 7-10 maps from company names to `Addresses`; the `Address` class is a trivial `record` (see Recipe 8.4) to keep the example short.

Example 7-10. main/src/main/java/structure/HashMapDemo.java

```
public class HashMapDemo {

  record Address(String city, String state){}

  public static void main(String[] argv) {

    // Construct and load the hash. This simulates loading a
    // database or reading from a file, or wherever the data is.

    Map<String,Address> map = new HashMap<>();

    // Populate it: the map maps from company name to Address.
    map.put("Adobe", new Address("Mountain View", "CA"));
    map.put("IBM", new Address("White Plains", "NY"));
    map.put("Learning Tree", new Address("Los Angeles", "CA"));
    map.put("Microsoft", new Address("Redmond", "WA"));
    map.put("O'Reilly", new Address("Sebastopol", "CA"));
    map.put("Rejminet Group", new Address("Toronto", "ON"));

    // Several versions of the "retrieval" phase.
    // Version 1: get one pair's value given its key
    // (presumably the key would really come from user input):
    System.out.println("Version 1");
    String queryString = "O'Reilly";
    System.out.println("You asked about " + queryString + ".");
    var result = map.get(queryString);
    System.out.println("They are located in: " + result);
    System.out.println();

    // Version 2: get ALL the keys and values
    // (maybe to print a report, or to save to disk)
    System.out.println("Version 2");
    for( String key : map.keySet()) {
      System.out.println("Key " + key +
        "; Value " + map.get(key));
    }
    System.out.println();

    // Version 3: Same but using a Map.Entry lambda
    System.out.println("Version 3");
    map.entrySet().forEach(mE ->
      System.out.println("Key " + mE.getKey()+
        "; Value " +mE.getValue()));
    System.out.println();
  }
}
```

For this version we used both a `for` loop and a `forEach()` loop; the latter uses the return from `entrySet()`, a set of `Map.Entry`, each of which contains one key and one value (this may be faster on large maps because it avoids going back into the map to

get the value each time through the loop). If you are modifying the map as you are going through it (e.g., removing elements), either inside the loop or in another thread, then these forms would fail with a `ConcurrentModificationException`. You then need to use an `Iterator` explicitly to control the loop:

```
// Version 2: get ALL the keys and values
// with concurrent modification
Iterator<String> it = map.keySet().iterator();
while (it.hasNext()) {
  String key = it.next();
  if (key.equals("Sun") || key.equals("Netscape")) {
    it.remove();
    continue;
  }
  System.out.println("Company " + key + "; " +
    "Address " + map.get(key));
}
```

A more functional (see Chapter 9) way of writing the removal, not involving explicit looping, would be this:

```
// Alternate 1 to just do the removals, without explicit looping
map.keySet().removeIf(key -> Set.of("Netscape", "Sun").contains(key));

// Alternate 2 to do the removals without an explicit loop
map .entrySet()
  .removeIf(entry -> Set.of("Netscape", "Sun")
    .contains(entry.getKey()));

// Show the results.
map.entrySet().forEach(System.out::println);
```

Both of the examples are from *javasrc/main/src/main/java/structure/HashMapWithRemoves.java*.

`HashMap` methods are not synchronized. The older and similar `Hashtable` methods are synchronized, for use with multiple threads, with some overhead.

7.10 Storing Strings in Properties and Preferences

Problem

You need to store keys and values that are both strings, possibly with persistence across runs of a program—for example, program customization.

Solution

Use a `java.util.prefs.Preferences` object or a `java.util.Properties` object.

Discussion

Here are two approaches to customization based on the user's environment. Java offers `Preferences` and `Properties` for cross-platform customizations.

Preferences

The `Preferences` class `java.util.prefs.Preferences` provides an easy-to-use mechanism for storing user customizations. While the API is totally platform-neutral, the prefs will be stored in a system-dependent way (which might mean dot files on Unix, a preferences file on the Mac, or the registry on Windows systems). This class provides a hierarchical set of nodes representing a user's preferences. Data is stored in the system-dependent storage format but can also be exported to or imported from an XML format. Example 7-11 is a simple demonstration of `Preferences`.

Example 7-11. main/src/main/java/structure/PrefsDemo.java

```
public class PrefsDemo {

  public static void main(String[] args) throws Exception {

    // Set up the Preferences for this application, by class.
    Preferences prefs = Preferences.userNodeForPackage(PrefsDemo.class);

    // Retrieve some preferences previously stored, with defaults in case
    // this is the first run.
    String text    = prefs.get("textFontName", "lucida-bright");
    String display = prefs.get("displayFontName", "lucida-blackletter");
    System.out.println(text);
    System.out.println(display);

    // Assume the user chose new preference values: Store them back.
    prefs.put("textFontName", "times-roman");
    prefs.put("displayFontName", "helvetica");

    // Toss in a couple more values for the curious who want to look
    // at how Preferences values are actually stored.
    Preferences child = prefs.node("a/b");
    child.putInt("meaning", 42);
    child.putDouble("pi", Math.PI);

    // And dump the subtree from our first node on down, in XML.
    prefs.exportSubtree(System.out);
```

```
  }
}
```

When you run the `PrefsDemo` program the first time, of course, it doesn't find any settings, so the calls to `preferences.get()` return the default values:

```
$ java -cp target/classes structure.PrefsDemo
lucida-bright
lucida-blackletter
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="structure">
      <map>
        <entry key="displayFontName" value="helvetica"/>
        <entry key="textFontName" value="times-roman"/>
      </map>
      <node name="a">
        <map/>
        <node name="b">
          <map>
            <entry key="meaning" value="42"/>
            <entry key="pi" value="3.141592653589793"/>
          </map>
        </node>
      </node>
    </node>
  </root>
</preferences>
```

On subsequent runs, it finds and returns the user-provided settings (I've elided the XML output from the second run because most of the XML output is the same):

```
> java structure.PrefsDemo
times-roman
helvetica
...
>
```

Properties

The `Properties` class is similar to a `HashMap` or `Hashtable` (it actually extends the latter) but with methods defined specifically for string storage and retrieval and for loading/saving. `Properties` objects are used throughout Java, for everything from setting the platform font names to customizing user applications into different `Locale` settings as part of internationalization and localization. When stored on disk, a `Properties` object looks just like a series of `name=value` assignments, with optional comments. Comments are added when you edit a `Properties` file by hand, ignored

when the Properties object reads itself, and lost when you ask the Properties object to save itself to disk. Here is an example of a Properties file that could be used to internationalize the menus in a GUI-based program:

```
# Default properties for MenuIntl
program.title=Demonstrate I18N (MenuIntl)
program.message=Welcome to an English-localized Java Program
#
# The File Menu
#
file.label=File Menu
file.new.label=New File
file.new.key=N
file.open.label=Open...
file.open.key=O
file.save.label=Save
file.save.key=S
file.exit.label=Exit
file.exit.key=Q
```

Here is another example, showing some personalization properties:

```
name=Ian Darwin
favorite_popsicle=cherry
favorite_rock group=Fleetwood Mac
favorite_programming_language=Java
pencil_color=green
```

A Properties object can be loaded from a file. The rules are flexible: either =, :, or spaces can be used after a key name and its values. Spaces after a nonspace character are ignored in the key. A backslash can be used to continue lines or to escape other characters. Comment lines may begin with either # or !. Thus, a Properties file containing the previous items, if prepared by hand, could look like this:

```
# Here is a list of properties
! first, my name
name Ian Darwin
favorite_popsicle = cherry
favorite_rock\ group \
 Fleetwood Mac
favorite_programming_language=Java
pencil_color green
```

Fortunately, when a Properties object writes itself to a file, it uses the following simple format:

```
key=value
```

Here is an example of a program that creates a Properties object and adds into it the list of companies and their locations from Recipe 7.9. It then loads additional properties from disk. To simplify the I/O processing, the program assumes that the Proper

ties file to be loaded is contained in the standard input, as would be done using a command-line redirection:

```
public class PropsCompanies {

  public static void main(String[] argv) throws java.io.IOException {

    Properties props = new Properties();

    // Get my data
    props.put("Adobe", "Mountain View, CA");
    props.put("IBM", "White Plains, NY");
    props.put("Learning Tree", "Los Angeles, CA");
    props.put("Microsoft", "Redmond, WA");
    props.put("Netscape", "Mountain View, CA");
    props.put("O'Reilly", "Sebastopol, CA");
    props.put("Sun", "Mountain View, CA");

    // Now load additional properties
    props.load(System.in);

    // List merged properties, using System.out
    props.list(System.out);
  }
}
```

Running it as:

```
java structure.PropsCompanies < PropsDemo.out
```

produces the following output in the file *PropsDemo.out*:

```
-- listing properties --
Sony=Japan
Sun=Mountain View, CA
IBM=White Plains, NY
Netscape=Mountain View, CA
Nippon_Kogaku=Japan
Acorn=United Kingdom
Adobe=Mountain View, CA
Ericsson=Sweden
O'Reilly & Associates=Sebastopol, CA
Learning Tree=Los Angeles, CA
```

Similarly to the `Set` examples, in the `HashMap` and `Properties` examples, the order in which the outputs appear is neither sorted nor in the order we put them in. The hashing classes and the `Properties` subclass make no claim about the order in which objects are retrieved. If you need them sorted, see Recipe 7.11.

As a convenient shortcut, my `FileProperties` class includes a constructor that takes a filename:

```
import com.darwinsys.util.FileProperties;
...
Properties p = new FileProperties("PropsDemo.out");
```

Note that constructing a `FileProperties` object causes it to be loaded, and therefore the constructor may throw a checked exception of class `IOException`.

7.11 Sorting a Collection

Problem

You put your data into a collection in random order or used a `Properties` object that doesn't preserve the order, and now you want it sorted.

Solution

Use the static method `Arrays.sort()` or `Collections.sort()`, optionally providing a `Comparator`.

Discussion

If your data is in an array, then you can sort it using the static `sort()` method of the `Arrays` utility class. If it is in a `Collection`, you can use the static `sort()` method of the `Collections` class. Here is a set of strings being sorted in place in an `Array`:

```
public class SortArray {
  public static void main(String[] unused) {
    String[] strings = {
      "painful",
      "mainly",
      "gaining",
      "raindrops"
    };
    Arrays.sort(strings);
    for (int i=0; i<strings.length; i++) {
      System.out.println(strings[i]);
    }
  }
}
```

What if the default sort order isn't what you want? Well, you can create an object that implements the `Comparator<T>` interface and pass that as the second argument to sort. Fortunately, for the most common ordering next to the default, you don't have to write your own: a public constant `String.CASE_INSENSITIVE_ORDER` can be passed as this second argument. The `String` class defines it as a `Comparator<String>` that orders `String` objects as by `compareToIgnoreCase`. But if you need something fancier, you probably need to write a `Comparator<T>`. In some cases you may be able

to use the `Comparator.comparing()` method and other static methods on `Comparator` to create a custom comparator without having to create a class. Suppose that, for some strange reason, you need to sort strings using all but the first character of the string. One way to do this would be to write this `Comparator<String>`:

```
/** Comparator for comparing strings ignoring first character.
 */
public class SubstringComparator implements Comparator<String> {
  @Override
  public int compare(String s1, String s2) {
    s1 = s1.substring(1);
    s2 = s2.substring(1);
    return s1.compareTo(s2);
    // or, more concisely:
    // return s1.substring(1).compareTo(s2.substring(1));
  }
}
```

Using it is just a matter of passing it as the `Comparator` argument to the correct form of `sort()`, as shown here:

```
public class SubstringComparatorDemo {
  public static void main(String[] unused) {
    String[] strings = {
      "painful",
      "mainly",
      "gaining",
      "raindrops"
    };
    Arrays.sort(strings);
    dump(strings, "Using Default Sort");
    Arrays.sort(strings, new SubstringComparator());
    dump(strings, "Using SubstringComparator");

    System.out.println("Functional approach:");
    Arrays.stream(strings)
      .sorted(Comparator.comparing(s->s.substring(1)))
      .forEach(System.out::println);
  }

  static void dump(String[] args, String title) {
    System.out.println(title);
    for (String s : args)
      System.out.println(s);
  }
}
```

Again, a more functional (see Chapter 9) way of writing this might be the following:

```
    System.out.println("Functional approach:");
    Arrays.stream(strings)
      .sorted(Comparator.comparing(s->s.substring(1)))
      .forEach(System.out::println);
```

Here is the output of running it:

```
$ java structure.SubstrCompDemo
Using Default Sort
gaining
mainly
painful
raindrops
Using SubstringComparator
raindrops
painful
gaining
mainly
```

And this is all as it should be.

On the other hand, you may be writing a class and want to build in the comparison functionality so that you don't always have to remember to pass the `Comparator` with it. In this case, the data class can directly implement the `java.lang.Comparable` interface, as is done by many classes in the standard API. These include the `String` class; the wrapper classes `Byte`, `Character`, `Double`, `Float`, `Long`, `Short`, and `Integer`; `BigInteger` and `BigDecimal` from `java.math`; most objects in the date/time API in `java.time`; and `java.text.CollationKey`. Arrays or `Collections` of these types can be sorted without providing a `Comparator`.

Classes that implement `Comparable` are said to have a natural ordering. The documentation strongly recommends that a class's natural ordering be consistent with its `equals()` method. It is consistent with `equals()` if and only if `e1.compareTo((Object)e2) == 0` has the same Boolean value as `e1.equals((Object)e2)` for every instance `e1` and `e2` of the given class. This means that if you implement `Comparable`, you should also implement `equals()`, and the logic of `equals()` should be consistent with the logic of the `compareTo()` method.

If you implement `equals()`, you should also implement `hashCode()` (as discussed in "hashCode() and equals()" on page 288). Here, for example, is part of the appointment class `Appt` from a hypothetical scheduling program. The class has a `LocalDate` date variable and a `LocalTime` time variable; the latter may be null (e.g., an all-day appointment or a to-do item); this complicates the `compareTo()` function a little:

```
// public class Appt implements Comparable {
  // Much code and variables omitted - see online version
  //-----------------------------------------------------------------
  //  METHODS - COMPARISON
  //-----------------------------------------------------------------
  /** compareTo method, from Comparable interface.
   * Compare this Appointment against another, for purposes of sorting.
   * <P>Only date and time, then text, participate, not repetition!
   * (Repetition has to do with recurring events, e.g.,
   * "Meeting every Tuesday at 9").
```

```
 * This method is consistent with equals().
 * @return -1 if this<a2, +1 if this>a2, else 0.
 */
@Override
public int compareTo(Appt a2) {
  // If dates not same, trigger on their comparison
  int dateComp = date.compareTo(a2.date);
  if (dateComp != 0)
    return dateComp;
  // Same date. If times not same, trigger on their comparison
  if (time != null && a2.time != null) {
    // Neither time is null
    int timeComp = time.compareTo(a2.time);
    if (timeComp != 0)
      return timeComp;
  } else /* At least one time is null */ {
    if (time == null && a2.time != null) {
      return -1; // All-day appts sort low to appear first
    } else if (time != null && a2.time == null)
      return +1;
      // else both have no time set, so carry on,
  }
  // Same date & time, trigger on text
  return text.compareTo(a2.text);
}

@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((date == null) ? 0 : date.hashCode());
  result = prime * result + ((text == null) ? 0 : text.hashCode());
  result = prime * result + ((time == null) ? 0 : time.hashCode());
  return result;
}

@Override
public boolean equals(Object o2) {
  if (this == o2)
    return true;
  if (o2.getClass() != Appt.class)
    return false;
  Appt a2 = (Appt) o2;
  if (!date.equals(a2.date))
    return false;
  if (time != null && !time.equals(a2.time))
    return false;
  return text.equals(a2.text);
}

/** Return a String representation of this Appt.
 * Output is intended for debugging, not presentation!
```

```
 */
@Override
public String toString() {
  var sb = new StringBuilder();
  sb.append(date).append(' ');
  if (time != null) {
    sb.append(time.getHour())
    .append(':')
    .append(time.getMinute())
    .append(' ');
  } else {
    sb.append("(All day)").append(' ');
  }
  sb.append(text).toString();
  return sb.toString();
}
```

If you're still confused between `Comparable` and `Comparator`, you're probably not alone. Table 7-3 summarizes the two comparison interfaces.

Table 7-3. `Comparable` compared with `Comparator`

Interface name	Description	Method(s)
`java.lang.Comparable<T>`	Provides a natural ordering to objects. Implemented in the class whose objects are being sorted.	`int compareTo(T o);`
`java.util.Comparator<T>`	Standalone strategy object provides full control over sorting objects of another class. Pass to `sort()` method or `Collection` constructor.	`int compare(T o1, T o2); boolean equals(T c2)`

Finally, not everything that requires order requires an explicit *sort* operation. You can avoid the overhead and elapsed time of an explicit sorting operation by ensuring that the data is in the correct order at all times, though this may or may not be faster overall, depending on your data and how you choose to keep it sorted. You can keep it sorted either manually or by using a `TreeSet` or a `TreeMap`. First, here is some code from a call tracking program that I first wrote on the very first public release of Java (the code has been modernized slightly!) to keep track of people I had had contact with and, in particular, those I needed to call back! Far less functional than a Rolodex, my `CallTrack` program maintained a list of people sorted by last name and first name. It also had the city, phone number, and email address of each person. Here is a very small portion of the code showing the "add person" operation:

```
/** Add one (new) Person to the list, keeping the list sorted. */
protected void add(Person p) {
  String lastName = p.getLastName();
  int i;
  // Find in "i" the position in the list where to insert this person
  for (i=0; i<usrList.size(); i++)
    if (lastName.compareTo((usrList.get(i)).getLastName()) <= 0)
```

```
            break; // If we don't break, OK, will insert at end of list.
        usrList.add(i, p);
```

This code uses a linear search, which was fine for the original application but could get very slow on large lists (it is *O(n)*). You'd need to use hashing or a binary search to find where to put the values on large lists.

If I were writing this code today, I would likely use a `TreeSet` (which keeps objects in order) or a `TreeMap` (which keeps the keys in order and maps from keys to values; the keys would be the name and the values would be the `Person` objects). Both insert the objects into a tree in the correct order (using the same logic as an explicit sort), so an `Iterator` that traverses the tree always returns the objects in sorted order. In addition, they have methods such as `headSet()` and `headMap()`, which give a new `Set` or `Map` of objects of the same class, containing the objects lexically before a given value. The `tailSet()` and `tailMap()` methods, similarly, return objects greater than a given value, and `subSet()` and `subMap()` return a range. The `first()` and `last()` methods retrieve the obvious components from the collection. The following program uses a `TreeSet` to sort some names:

```
// A TreeSet keeps objects in sorted order. Use a Comparator
// published by String for case-insensitive sorting order.
TreeSet<String> theSet = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
theSet.add("Gosling");
theSet.add("da Vinci");
theSet.add("van Gogh");
theSet.add("Java To Go");
theSet.add("Vanguard");
theSet.add("Darwin");
theSet.add("Darwin");  // TreeSet is Set, ignores duplicates.

System.out.printf("Our set contains %d elements", theSet.size());

// Since it is sorted we can easily get various subsets
System.out.println("Lowest (alphabetically) is " + theSet.first());

// Print how many elements are greater than "k"
// Should be 2 - "van Gogh" and "Vanguard"
System.out.println(theSet.tailSet("k").toArray().length +
  " elements higher than \"k\"");

// Print the whole list in sorted order
System.out.println("Sorted list:");
theSet.forEach(name -> System.out.println(name));
```

One last point to note is that if you have a `Hashtable` or `HashMap`, you can convert it to a `TreeMap`, and therefore get it sorted, just by passing it to the `TreeMap` constructor:

```
TreeMap sorted = new TreeMap(unsortedHashMap);
```

7.12 Finding an Object in a Collection

Problem

You need to see whether a given collection contains a particular value.

Solution

Ask the collection if it contains an object of the given value.

Discussion

If you have created the contents of a collection, you probably know what is in it and what is not. But if the collection is prepared by another part of a large application, or even if you've just been putting objects into it and now need to find out if a given value was found, this recipe's for you. There are various methods available, depending on which `Collection` class you have. The methods in Table 7-4 can be used.

Table 7-4. Finding objects in a collection

Method(s)	Meaning	From interface	Implementing classes
`binarySearch()`	Fairly fast search	`Collection`	`Arrays`, `Collections`
`contains()`	Search	`Collection`	`ArrayList`, `HashSet`, `Hashtable`, `LinkList`, `Properties`, `Vector`
`containsKey()`, `containsValue()`	Checks if the collection contains the object as a `Key` or as a `Value`	`Map`	`HashMap`, `Hashtable`, `Properties`, `TreeMap`
`indexOf()`	Returns location where object is found	`Collection`	`ArrayList`, `LinkedList`, `List`, `Stack`, `Vector`
`search()`	Search	N/A	`Stack`

The methods whose names start with `contains` will (probably) use a linear search if the collection is a `Collection` (`List`, `Set`) but will be faster if the collection is hashed (`HashSet`, `HashMap`).

The next example plays a game of "find the hidden number" (or "needle in a haystack"): the player only gets one guess to see if their number is in the haystack; this trivial version always guesses the number 1,999. If the player loses, the game must report the nearest number to the player's guess, so we couldn't just use a `List` and call `List.contains()`. Instead, the numbers to look through are stored in an array. As games go, it's fairly pathetic: the computer plays against itself, so you probably know

who's going to win. I wrote it that way so I would know that the data array contains valid numbers.

The interesting part is not the generation of the random numbers (discussed in Recipe 5.9). The array to be used with `Arrays.binarySearch()` must be in sorted order, but since we just filled it with random numbers, it isn't initially sorted. Hence, we call `Arrays.sort()` on the array. Then we are in a position to call `Arrays.binarySearch()`, passing in the array and the value to look for. If you run the program with a number, it runs that many games and reports on how it fared overall. If you don't bother, it plays only one game:

```
public class ArrayHunt  {
  /** the maximum (and actual) number of random ints to allocate */
  protected final static int MAX    = 4000;
  /** the value to look for */
  protected final static int NEEDLE = 1999;
  int[] haystack;
  Random r;

  public static void main(String[] argv) {
    ArrayHunt h = new ArrayHunt();
    if (argv.length == 0)
      h.play();
    else {
      int won = 0;
      int games = Integer.parseInt(argv[0]);
      for (int i=0; i<games; i++)
        if (h.play())
          ++won;
      System.out.println("Computer won " + won +
        " out of " + games + ".");
    }
  }

  /** Construct the hunting ground */
  public ArrayHunt() {
    haystack = new int[MAX];
    r = new Random();
  }

  /** Play one game. */
  public boolean play() {
    int i;

    // Fill the array with random data (hay?)
    for (i=0; i<MAX; i++) {
      haystack[i] = (int)(r.nextFloat() * MAX);
    }

    // Precondition for binary search is that data be sorted!
    Arrays.sort(haystack);
```

```
        // Look for needle in haystack
        i = Arrays.binarySearch(haystack, NEEDLE);

        if (i >= 0) {    // Found it, we win.
          System.out.println("Value " + NEEDLE +
            " occurs at haystack[" + i + "]");
          return true;
        } else {    // Not found, we lose.
          System.out.println("Value " + NEEDLE +
            " does not occur in haystack; nearest value is " +
            haystack[-(i+2)] + " (found at " + -(i+2) + ")");
          return false;
        }
      }
    }
```

`Collections.binarySearch()` works almost exactly the same way, except it looks in a `Collection`, which must be sorted (presumably using `Collections.sort`, as discussed in Recipe 7.11).

7.13 Converting Between Collections and Arrays

Problem

You have a `Collection` but you need a Java language array.

Solution

Use the `Collection` method `toArray()`.

Discussion

If you have an `ArrayList` or other `Collection` and you need an array (either to deal with some other API, or just for speed), you can get it just by calling the `Collection`'s `toArray()` method. With no arguments, you get an array whose type is `Object[]`. You can optionally provide an array argument, which is used for two purposes:

- The type of the array argument determines the type of array returned.
- If the array is big enough (and you can ensure that it is by allocating the array based on the `Collection`'s `size()` method), then this array is filled and returned. If the array is not big enough, a new array is allocated instead. If you provide an array and objects in the `Collection` cannot be cast to this type, then you will get an `ArrayStoreException`.

Example 7-12 shows code for converting an `ArrayList` to an array of type `Object`.

Example 7-12. main/src/main/java/structure/ToArray.java

```
List<String> list = new ArrayList<>();
list.add("Blobbo");
list.add("Cracked");
list.add("Dumbo");

// Convert a collection to Object[], which can store objects
// of any type.
Object[] ol = list.toArray();
System.out.println("Array of Object has length " + ol.length);

String[] sl = (String[]) list.toArray(new String[0]);
System.out.println("Array of String has length " + sl.length);
```

7.14 Making Your Own Data Structures Iterable

Problem

You have written your own data structure, and you want to publish the data to be iterable so it can be used in the for-each loop.

Solution

Make your data class `Iterable`: this interace has only one method, `iterator()`, which will return an instance of your own `Iterator`. Just implement (or provide an inner class that implements) the `Iterator` interface.

Discussion

To be usable as the container object in the modern Java for-each loop, your data class must implement `Iterable`, a simple interface with one method, `Iterator<T> iterator()`. Whether you use this interface or want to use the older `Iterator` interface directly, the way to make data from one part of your program available in a storage-independent way to other parts of the code is to generate an `Iterator`. Here is a short program that constructs, upon request, an `Iterator` for some data that it is storing—in this case, in an array. The `Iterator` interface has only three methods—`hasNext()`, `next()`, and `remove()`—demonstrated in Example 7-13.

Example 7-13. main/src/main/java/structure/IterableDemo

```
public class IterableDemo {

  /** Demo implements Iterable, meaning it must provide an Iterator,
   * and that it can be used in a foreach loop.
   */
  static class Demo implements Iterable<String> {
```

```
    // Simple demo: use array instead of inventing new data structure
    String[] data = { "One", "Two", "Three"};

    /** This is the Iterator that makes it all happen */
    class DemoIterator implements Iterator<String> {
      int i = 0;

      /**
       * Tell if there are any more elements.
       * @return true if next() will succeed, false otherwise
       */
      public boolean hasNext() {
        return i < data.length;
      }

      /** @return the next element from the data */
      public String next() {
        return data[i++];
      }

      /** Remove the object that next() just returned.
       * An Iterator is not required to support this method, and we don't.
       * @throws UnsupportedOperationException unconditionally
       */
      public void remove() {
        throw new UnsupportedOperationException("remove");
      }
    }

    /** Method by which the Demo class makes its iterator available */
    public Iterator<String> iterator() {
      return new DemoIterator();
    }
  }

  public static void main(String[] args) {
    Demo demo = new Demo();
    for (String s : demo) {
      System.out.println(s);
    }
  }
}
```

The comments on the `remove()` method remind me of an interesting point. This interface introduces `java.util`'s attempt at something Java doesn't really have, the *optional method*. Because there is no syntax for this, and they didn't want to introduce any new syntax, the developers of the Collections Framework decided on an implementation using existing syntax. Optional methods that are not implemented will usually throw an `UnsupportedOperationException` if they ever get called. My

`remove()` method does just that. `UnsupportedOperationException` is subclassed from `RuntimeException`, so it is not required to be declared or caught.

This code is simplistic, but it does show the syntax and demonstrates how the `Iterator` interface works. In real code, the `Iterator` and the data are usually separate objects (the `Iterator` might be an inner class from the data store class, as shown in Example 7-13). I show this simple example of the internals of an `Iterator` so that readers understand both how it works and how one could be provided for a more sophisticated data structure, should the need arise.

The `Iterable` interface has only one nondefault method, `iterator()`, which must provide an `Iterator` for objects of the given type. Because the `ArrayIterator` class implements this as well, we can use an object of type `ArrayIterator` in a for-each loop, as in Example 7-14.

Example 7-14. main/src/main/java/structure/ArrayIteratorDemo.java

```
package structure;

import com.darwinsys.util.ArrayIterator;

public class ArrayIteratorDemo {

  private final static String[] names = {
    "rose", "petunia", "tulip"
  };

  public static void main(String[] args) {
    ArrayIterator<String> arrayIterator = new ArrayIterator<>(names);

    System.out.println("Java 5, 6 way");
    for (String s : arrayIterator) {
      System.out.println(s);
    }

    System.out.println("Java 8 ways");
    arrayIterator.forEach(s->System.out.println(s));
    arrayIterator.forEach(System.out::println);
  }
}
```

Iterable.forEach() 8

Java 8 added a `forEach` method to the `Iterable` interface, a *default method* (discussed in the introdcution to Chapter 9) that you don't have to write. Thus, without changing the `ArrayIterator`, after moving to Java 8 we can use the newest-style loop, `Iterator.forEach(Consumer)`, with a lambda expression (see Chapter 9) to print each element (see Example 7-14).

7.15 Multidimensional Structures

Problem

You need a multidimensional array or `List`. This is a common requirement in mathematical and data science applications, among others.

Solution

No problem. Java supports this.

Discussion

As mentioned in Recipe 7.1, Java arrays can hold any reference type. Since an array is a reference type, it follows that you can have arrays of arrays, or *multidimensional* arrays. And since each array has its own length attribute, the columns of a two-dimensional array, for example, do not all have to be the same length (see Figure 7-3).

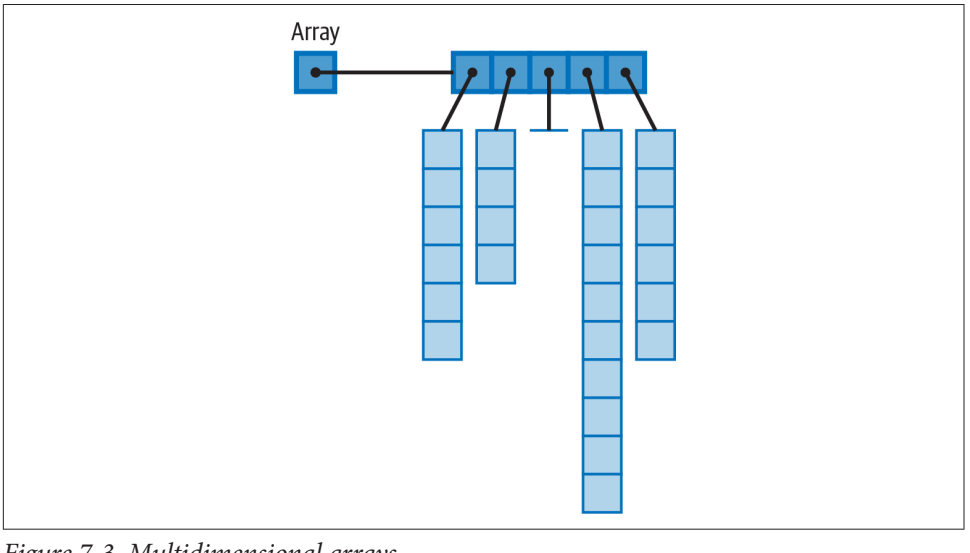


Figure 7-3. Multidimensional arrays

Here is code to allocate a couple of two-dimensional arrays, one using a loop and the other using an initializer. Both are selectively printed:

```
public class ArrayTwoDObjects {

  /** Return list of subscript names (unrealistic; just for demo). */
  public static String[][] computedArray() {
    String info[][];
    info = new String[10][10];
```

```
    for (int i=0; i < info.length; i++) {
      for (int j = 0; j < info[i].length; j++) {
        info[i][j] = "String[" + i + "," + j + "]";
      }
    }
    return info;
  }

  static String[][] inittedArray = {
    { "One", "Two" },
    { "Buckle my shoe" },
  };

  /** Run the initialization method and print part of the results */
  public static void main(String[] args) {
    print("from computedArray", computedArray());
    print("from initted array", inittedArray);
  }

  /** Print selected elements from the 2D array */
  public static void print(String tag, String[][] array) {
    System.out.println("Array " + tag + " is of length " + array.length);
    for (int i=0; i < 2; i++) {
      System.out.println("Row " + i + " is " + array[i].length);
      for (int j = 0; j < Math.min(array[i].length, 3); j++) {
        System.out.printf("Array[%d][%d] = array[%d][%d]}", i, j, i, j);
      }
    }
  }
}
```

Running it produces this output:

```
> java structure.ArrayTwoDObjects
Array from computedArray is of length 10
Row 0 is 10
Array[0][0] = String[0,0]
Array[0][1] = String[0,1]
Array[0][2] = String[0,2]
Row 1 is 10
Array[1][0] = String[1,0]
Array[1][1] = String[1,1]
Array[1][2] = String[1,2]
Array from initted array is of length 2
Row 0 is 2
Array[0][0] = One
Array[0][1] = Two
Row 1 is 1
Array[1][0] = Buckle my shoe
```

The same kind of logic can be applied to any of the `Collections`. You could have an `ArrayList` of `ArrayLists`, or a `Vector` of linked lists, or whatever your little heart desires.

As Figure 7-3 shows, it is not necessary for the array to be regular (i.e., it's possible for each column of the 2D array to have a different height). This is different from the conventional view of a matrix, which is generally assumed to be rectangular. That is why I used `array[0].length` for the length of the first column in the code example.

CHAPTER 8

Object-Oriented Techniques

8.0 Introduction

Java is an object-oriented (OO) language in the tradition of Simula-67, SmallTalk, and C++. It borrows syntax from C++ and ideas from SmallTalk. The Java API has been designed and built on the OO model and following many OO design patterns. The idea of "design patterns" originated with Christopher Alexander and colleagues at UC Berkeley's Faculty of Architecture, as a way of teaching budding architects how to better design homes and offices.[1] In the 1980s and early 1990s, computer science departments published catalogs of design patterns in computing. The 1995 book *Design Patterns* (by Erich Gamma et al., Addison-Wesley, often called the "Gang of Four" book [GoF], after the four authors) catalogued some two dozen of the best. These design patterns, such as Proxy, Strategy, and Delegate, are used throughout modern development; an understanding of these patterns will help you better understand the use of the Java API and improve the design of your own classes.

Most of the GoF book examples were originally written in C++. For Java developers, the pieces I wrote for Oracle's *Java Magazine* include articles on half a dozen or so of these patterns. The book *Head First Design Patterns* by Eric Freeman and Elisabeth Robson (O'Reilly) describes all 23 GoF patterns, with examples in Java in a very entertaining and memorable fashion.

1 See Alexander et al., *A Pattern Language*, Oxford University Press, 1977.

Advice, or Mantras

I could give you any number of short bits of advice on object-oriented programming. A few recurring themes arise when learning the basics of Java, and I suggest reviewing them when learning advanced Java.

Use the API

I can't say this often enough. A lot of the things you need to do have already been done by the good folks who develop the standard Java library (and third-party libraries). And this grows with every release. Learning the API well is a good foundation for avoiding that deadly "reinventing the flat tire" syndrome—coming up with a second-rate equivalent of a first-rate product that was available to you the whole time. In fact, part of this book's mission is to prevent you from reinventing what's already there. One example of this is the Collections API in java.util, discussed in Chapter 7. The Collections API has a high degree of generality and regularity, so there is often no need to invent your own data structuring code.

Learn exceptions to the rule

There are a few exceptions to the rule of using the API: the `clone()` method in `java.lang.Object` should generally *not* be used. If you need to copy an object, just write a copy method, or a *copy constructor*. For `Lists`, use the `ArrayList(List)` copy constructor. If you need to copy an array, use the static `Arrays.copyOf()` method. Joshua Bloch's arguments against the `clone()` method in the book *Effective Java* (Addison-Wesley) are persuasive and should be read by any dedicated Java programmer. While you're at it, read that whole book.

Another exception is the `finalize()` method in `java.lang.Object()`. Don't use it. It has been deprecated since Java 9 and ultimately marked for removal. It isn't guaranteed to be invoked; but because it might get invoked, it will cause your dead objects not to be garbage collected, resulting in a memory leak. If you need some kind of cleanup, take responsibility for defining a method and invoking it before you let any object of that class go out of reference. You might call such a method `cleanUp()`. For application-level cleanup, see the recipe for shutdown hooks on my website.

Generalize

There is a trade-off between generality (and the resulting reusability), which is emphasized here, and the convenience of application specificity. If you're writing one small part of a very large application designed according to OO design techniques, you'll have in mind a specific set of use cases. On the other hand, if you're writing toolkit-style code, you should write classes with few assumptions about how they'll be used. Making code easy to use from a variety of programs is the route to writing reusable code.

Read and write Javadoc

You've no doubt looked at the Java online documentation in a browser, in part because I just told you to learn the API well. Do you think Sun/Oracle hired millions of tech writers to produce all that documentation? No. That documentation exists because the developers of the API took the time to write Javadoc comments, those funny `/**` comments you've seen in code. So, one more bit of advice: use Javadoc. The standard JDK provides a good standard mechanism for API documentation. And use it as you write the code—don't think you'll come back and write it in later. Tomorrow never arrives.

See Recipe 1.7 for details on using Javadoc.

Use subclassing and delegation

Use subclassing. But don't overuse subclassing. It is one of the best ways to not only avoid code duplication but also to develop software that works. See any number of good books on the topic of object-oriented design and programming for more details.

There are several alternatives to subclassing. One alternative is delegation. Think about "is a" versus "has a." For example, instead of subclassing `NameAndAddress` to make `BusinessPartner` and `Customer`, make `BusinessPartner` and `Customer` have instances of `NameAndAddress`. That is a clearer structure; having `BusinessPartner` *be a* `NameAndAddress` just because the partner *has a* name and address would not necessarily make sense. Delegation also makes it easier for a `Customer` to have both a billing address and a shipping address.

Another alternative is aspect-oriented programming (AOP), which allows you to bolt on extra functionality from the outside of your classes. AOP is provided by the Jakarta EE using EJB interception and by the Spring Framework AOP mechanism, both of which fall outside the scope of this book.

Use design patterns

In the introduction to this chapter, I mentioned design patterns as one of the important topics in object-oriented programming. Popularized by the Gang of Four (GoF) book and *Head First Design Patterns*, the idea provides a powerful catalog of things that programmers often reinvent. A design pattern provides a statement of a problem and its solution(s), rather like the present book, but generally at a higher level of abstraction. It is as important for giving a standard vocabulary of design as it is for its clear explanations of how the basic patterns work and how they can be implemented.

Table 8-1 shows some example uses of design patterns in the standard API.

Table 8-1. Design patterns in the JavaSE API

Pattern name	Meaning	Examples in Java API
Command	Encapsulate requests, allowing queues of requests, undoable operations, etc.	`javax.swing.Action`; `javax.swing.undo.UndoableEdit`
Decorator	One class decorates another	Swing `Borders`
Factory Method	One class makes up instances for you, controlled by subclasses	`getInstance` (in `Calendar`, `Format`, `Locale`…); `SocketFactory`; RMI `InitialContext`
Iterator	Loop over all elements in a collection, visiting each exactly once	`Iterator`; older `Enumeration`; `java.sql.ResultSet`
Model-View-Controller	Model represents data; View is what the user sees; Controller responds to user requests	`ActionListener` and friends; `Observer`/`Observable`; used internally by all visible Swing components
Proxy	One object stands in for another	RMI, AOP, Dynamic Proxy
Singleton	Only one instance may exist	`java.lang.Runtime`, `java.awt.Toolkit`

I have written articles on the State, Proxy, Command, Decorator, and Visitor patterns for Oracle's *Java Magazine*.

8.1 Object Methods: Formatting Objects with toString(), Comparing with Equals

Problem

You want your objects to have a useful default formating capability and to behave themselves when placed in `Collections` classes.

Solution

There are three important overridable methods inherited from `java.lang.Object`; of these, `toString()` provides default formatting, while `equals()` and `hashCode()` provide equality testing and efficient usage in `Map` implementations. Some others, such as `clone()` and `finalize()` and the `wait()` versions, are not recommended for general use.

Discussion

toString()

Whenever you pass an object to `System.out.println()` or any equivalent method, or involve it in string concatenation, Java automatically calls its `toString()` method. Java knows that every object has a `toString()` method because `java.lang.Object` has one, and all classes are ultimately subclasses of `Object`. The default implementa-

tion, in `java.lang.Object`, is neither pretty nor interesting: it just prints the class name, an @ sign, and the object's `hashCode()` value. For example, if you run the code:

```
public class ToStringWithout {
    int x, y;

    /** Simple constructor */
    public ToStringWithout(int anX, int aY) {
        x = anX; y = aY;
    }

    /** Main just creates and prints an object */
    public static void main(String[] args) {
        System.out.println(new ToStringWithout(42, 86));
    }
}
```

you might see this uninformative output:

```
ToStringWithout@990c747b
```

If you want to make objects of your class print better, provide an implementation of `toString()` that prints the class name and some of the important states; in fact, this is good practice for all but the most trivial classes. This gives you formatting control in `println()`, in debuggers, and anywhere your objects get referred to in a `String` context. Here is the previous program rewritten with a `toString()` method:

```
public class ToStringWith {
  int x, y;

  /** Simple constructor */
  public ToStringWith(int anX, int aY) {
    x = anX; y = aY;
  }

  @Override
  public String toString() {
    return "ToStringWith[" + x + "," + y + "]";
  }

  /** Main just creates and prints an object */
  public static void main(String[] args) {
    System.out.println(new ToStringWith(42, 86));
  }
}
```

This version produces the more useful output:

```
ToStringWith[42,86]
```

This example uses `String` concatenation, but you may also want to use `String.format()` or `StringBuilder`; see Chapter 3.

Also note that the Java `record` type provides a default `toString()` method automatically (see "Java record type 16" on page 297).

hashCode() and equals()

To ensure your classes work correctly when any client code calls `equals()` or when these objects are stored in `Map` or other `Collections` classes, outfit your class with `equals()` and `hashCode()` methods.

How do you determine equality? For arithmetic or Boolean operands, the answer is simple: you test with the equals operator (==). For object references, though, Java provides both == and the `equals()` method inherited from `java.lang.Object`. The == operator can be confusing because it simply compares two object references to see if they refer to the same object. This is not the same as comparing the values of the objects themselves.

The inherited `equals()` method is also not as useful as you might imagine. Some people seem to start their lives as Java developers thinking that the default `equals()` magically does some kind of detailed, field-by-field, or even binary comparison of objects. But it does *not* compare fields! It just does the simplest possible thing: it returns the value of an == comparison on the two objects involved. So, for any *value classes* you write, you probably have to write an `equals` method.[2] Note that both the `equals` and `hashCode` methods are used by `Maps` or hashes (such as `HashMap`; see Recipe 7.9). So if you think somebody using your class might want to use instances as keys in a `Map`, or even compare your objects, you owe it to them (and to yourself!) to implement both `equals()` and `hashCode()` and to implement them properly.[3]

Most IDEs know how to generate usable `equals()` and `hashCode()` methods, but it's worth your while to understand what these are doing, for the occasional case where you need to tweak the generated code. Most have some kind of `Generate hashCode() and equals()` menu item that will only do both at the same time, not let you generate `equals()` without `hashCode()` or vice versa.

Here are the rules for a correct `equals()` method:

It is reflexive
: `x.equals(x)` must be true.

2 A value class is one used mainly to hold state, rather than logic: a `Person` is a value class, whereas `java.lang.Math` is not. Many classes are somewhere in between.

3 Andrew Slice points out that this reminds him of a common quote: "It's going to be the next guy's problem, and the next guy is probably you."

It is symmetrical
: `x.equals(y)` must be true if and only if `y.equals(x)` is also true.

It is transitive
: If `x.equals(y)` is true and `y.equals(z)` is true, then `x.equals(z)` must also be true.

It is idempotent (repeatable)
: Multiple calls on `x.equals(y)` return the same value (unless state values used in the comparison are changed, as by calling a set method).

It is cautious
: `x.equals(null)` must return false rather than accidentally throwing a `NullPointerException`.

In addition, beware of one common mistake: the argument to `equals()` must be declared as `java.lang.Object`, not the class it is in; this is so that polymorphism will work correctly (some classes may not have an `equals()` method of their own). To prevent this mistake, the `@Override` annotation is usually added to the `equals()` override, as mentioned in Recipe 1.8.

Here is a class that endeavors to implement these rules:

```
public class EqualsDemo {
    private int int1;
    private SomeClass obj1;

    /** Constructor */
    public EqualsDemo(int i, SomeClass o) {
        int1 = i;
        if (o == null) {
            throw new IllegalArgumentException("Data Object may not be null");
        }
        obj1 = o;
    }

    /** Default Constructor */
    public EqualsDemo() {
        this(0, new SomeClass());
    }

    /** Demonstration "equals" method */
    @Override
    public boolean equals(Object o) {
        if (o == this)                          ❶
            return true;

        if (o == null)                          ❷
            return false;
```

```
        // Of the correct class?
        if (o.getClass() != EqualsDemo.class) ❸
            return false;

        EqualsDemo other = (EqualsDemo)o; // OK, cast to this class

        // compare field-by-field              ❹
        if (int1 != other.int1)           // compare primitives directly
            return false;
        if (!obj1.equals(other.obj1))     // compare objects using their equals
            return false;
        return true;
    }

    // ...
```

❶ Optimization: if same object, true by definition.

❷ If other object null, false by definition.

❸ Compare class descriptors using `!=`; see following paragraph.

❹ Optimization: compare primitives first. May or may not be worthwhile; may be better to order by those most likely to differ—depends on the data and the usage.

Another common mistake to avoid: note the use of class descriptor equality (i.e., `o.getClass() != EqualsDemo.class`) to ensure the correct class, rather than via `instanceof`, as is sometimes erroneously done. The reflexive requirement of the `equals()` method contract pretty much makes it impossible to compare a subclass with a superclass correctly, so we now use class equality (see Chapter 17 for details on the class descriptor).

Here is a basic JUnit test (see Recipe 2.9) for the `EqualsDemo` class:

```
/** Some JUnit test cases for EqualsDemo.
 * Writing a full set is left as "an exercise for the reader".
 */
class EqualsDemoTest {

  /** an object being tested */
  EqualsDemo d1;
  /** another object being tested */
  EqualsDemo d2;

  /** Method to be invoked before each test method */
  @BeforeEach
  void setUp() {
    d1 = new EqualsDemo();
    d2 = new EqualsDemo();
  }
```

```
    @Test
    void symmetry() {
      assertEquals(d1, d1);
    }

    @Test
    void symmetric() {
      assertTrue(d1.equals(d2) == d2.equals(d1));
    }

    @Test
    void caution() {
      assertNotEquals(null, d1);
    }
  }
```

With all that testing, what could go wrong? Well, some things still need care. What if the object is a *subclass* of `EqualsDemo`? We should test that it returns false in this case.

What else could go wrong? Well, what if either `obj1` or `other.obj1` is null? You might have just earned a nice shiny new `NullPointerException`. So you also need to test for any possible null values. Good constructors can avoid these `NullPointerExceptions`, as I've tried to do in `EqualsDemo`, or else test for them explicitly.

Finally, you should never override `equals()` without also overriding `hashCode()`, and the same fields must take part in both computations.

hashCode()

The `hashCode()` method is supposed to return an `int` that should uniquely identify any set of values in objects of its class.

A properly written `hashCode()` method will follow these rules:

It is repeatable
: `hashCode(x)` must return the same `int` when called repeatedly, unless set methods have been called.

It is consistent with equality
: If `x.equals(y)`, then `x.hashCode()` must == `y.hashCode()`.

Distinct objects should produce distinct `hashCodes`
: If `!x.equals(y)`, it is not required that `x.hashCode()` != `y.hashCode()`, but doing so may improve performance of hash tables (i.e., hashes may call `hashCode()` before `equals()`).

The default `hashCode()` on the standard JDK returns a machine address, which conforms to the first rule. Conformance to the second and third rules depends, in part,

on your `equals()` method. Here is a program that prints the hashcodes of a small handful of objects:

```
public class PrintHashCodes {

  /** Some objects to hashCode() on */
  protected static Object[] data = {
    new PrintHashCodes(),
    new java.awt.Color(0x44, 0x88, 0xcc),
    new SomeClass()
  };

  public static void main(String[] args) {
    System.out.println("About to hashCode " + data.length + " objects.");
    for (int i=0; i<data.length; i++) {
      System.out.println(data[i].toString() + " --> " +
        data[i].hashCode());
    }
    System.out.println("All done.");
  }
}
```

What does it print?

```
> javac -d . oo/PrintHashCodes.java
> java oo.PrintHashCodes
About to hashCode 3 objects.
PrintHashCodes@982741a0 --> -1742257760
java.awt.Color[r=68,g=136,b=204] --> -12285748
SomeClass@860b41ad --> -2046082643
All done.
>
```

The hashcode value for the `Color` object is interesting. It is actually computed as something like this:

```
alpha<<24 + r<<16 + g<<8 + b
```

In this formula, `r`, `g`, and `b` are the red, green, and blue components, respectively, and `alpha` is the transparency. Each of these quantities is stored in 8 bits of a 32-bit integer. If the alpha value is greater than 128, the high bit in this word—having been set by shifting into the sign bit of the word—causes the integer value to appear negative when printed as a signed integer. Hashcode values are of type `int`, so they are allowed to be negative.

Again, the `record` type provides `equals()` and `hashCode()`. These make all fields take part, so you might want to override them if that's not appropriate.

Difficulties and Alternatives to Clone

The `java.util.Observable` class (designed to implement the Model-View-Controller pattern with AWT or Swing applications) contains a private `Vector` but no clone method to deep-clone it. Thus, `Observable` objects cannot safely be cloned, ever!

This and several other issues around `clone()`—such as the uncertainty of whether a given `clone()` implementation is deep or shallow—suggest that `clone()` was not as well thought out as it might be. An alternative is simply to provide a copy constructor or similar method:

```
public class CopyConstructorDemo {
  public static void main(String[] args) {
    CopyConstructorDemo object1 = new CopyConstructorDemo(123, "Hello");
    CopyConstructorDemo object2 = new CopyConstructorDemo(object1);
    if (!object1.equals(object2)) {
      System.out.println("Something is terribly wrong...");
    }
    System.out.println("All done.");
  }

  private int number;
  private String name;

  /** Default constructor */
  public CopyConstructorDemo()  {
  }

  /** Normal constructor */
  public CopyConstructorDemo(int number, String name)  {
    this.number = number;
    this.name = name;
  }

  /** Copy Constructor */
  public CopyConstructorDemo(CopyConstructorDemo other)  {
    this.number = other.number;
    this.name = other.name;
  }
  // hashCode() and equals() not shown
```

8.2 Constructor Simplification: Statements Before super(...) 22P

Problem

In writing a constructor, you need to compute or test some values before calling the superclass constructor.

Solution

Use a Java 22 preview feature that allows nonfield computations before the call to `super()`.

Discussion

For most of its history, Java would not allow any code in a constructor before the call to `super()`. This is necessary for object construction to work properly as a chained process, to ensure that superclass fields are initialized before the subclass constructor can access them. However, the restriction was perhaps too strict. It is quite possible to do some operations that cannot affect the outcome of the superclass constructor. In Java 22 preview, JEP 447 "Statements before super()" allows these. Example 8-1 shows code that will only work on Java 22.

Example 8-1. main/src/main/java//oo/StatementsBeforeSuper.java

```
class One {
  int dom;
  One(int dom) {
    if (dom <= 0 || dom > 31)
      throw new IllegalArgumentException(
        "Day of Month out of range in call to One()");
    this.dom = dom;
  }
}

class Two extends One {
  Two() {
    var x = LocalDate.now().getDayOfMonth();
    super(x);
  }
}

void main() {
  var o = new Two();
  System.out.println("Day of Month is " + o.dom);
}
```

Running this example on a Java 22 JDK results in the following:

```
$ javac --enable-preview --source 22 oo/StatementsBeforeSuper.java
$ java --enable-preview -cp oo StatementsBeforeSuper
Day of Month is 5
$ date
Sun November  5 14:51:29 EDT 2023
$
```

Code before `super()` is considered to be in a "pre-construction context," and cannot access instance methods. The complexities thereof are explored in JEP 447, which introduced this enhancement into Java 22.

8.3 Using Inner Classes

Problem

You need to write a private class, or a class to be used in one other class at most.

Solution

Use a nonpublic class or an inner class.

Discussion

A nonpublic class can be written as part of another class's source file, but not inside that class. An inner class is Java terminology for a class defined inside another class. Inner classes were first popularized with early Java for use as event handlers for GUI applications, but they have a much wider application.

Inner classes can, in fact, be constructed in several contexts. An inner class defined as a member of a class can be instantiated anywhere in that class. An inner class defined inside a method can be referred to later only in the same method. Inner classes can also be named or anonymous. A named inner class has a full name that is compiler dependent; the standard JVM uses a name like `MainClass$InnerClass` for the resulting file. An anonymous inner class, similarly, has a compiler-dependent name; the JVM uses `MainClass$1`, `MainClass$2`, and so on.

These classes cannot be instantiated in any other context; any explicit attempt to refer to, say, `OtherMainClass$InnerClass`, is caught at compile time:

main/src/main/java/oo/AllClasses.java

```
public class AllClasses {
  public class Data {                    ❶
    int x;
    int y;
  }
```

```
    public void getResults() {
      JButton b = new JButton("Press me");
      b.addActionListener(new ActionListener() { ❷
        public void actionPerformed(ActionEvent evt) {
          Data loc = new Data();
          loc.x = ((Component)evt.getSource()).getX();
          loc.x = ((Component)evt.getSource()).getY();
          System.out.println("Thanks for pressing me");
        }
      });
    }
}

/** Class contained in same file as AllClasses, but can be used
 * (with a warning) in other contexts.
 */
class AnotherClass {                                    ❸
  // methods and fields here...
  AnotherClass() {
    // Inner class from above cannot be used here, of course
    // Data d = new Data();  // EXPECT COMPILE ERROR
  }
}
```

❶ This is an inner class, which can be used anywhere in class `AllClasses`.

❷ This shows the anonymous inner class syntax, which uses `new` with a type followed by `(){`, a class body, and `}`. The compiler will assign a name; the class will extend or implement the given type, as appropriate.

❸ This is a nonpublic class; it can be used in the main class and (with warning) in other classes.

One issue is that the inner class retains a reference to the outer class. This is important if, for example, the inner class is an iterator that will be in use for a long time. If you want to avoid memory leaks if the inner class will be held for a longer time than the outer, you can make the inner class `static`. In that case it cannot, of course, access instance (nonstatic) methods or fields in the outer class.

Inner classes implementing a single-method interface—including the `ActionListener` in the preceding example—can be written in a much more concise fashion as lambda expressions (see Chapter 9).

8.4 Simplifying Data Objects with Records (or Lombok)

Problem

You spend time writing data classes that are Plain Old Java Objects (POJO), with boilerplate code such as setters and getters, `equals()`, and `toString()`.

Solution

In Java 16+, use the `record` data type, which generates 100% of the boilerplate methods for you. On older versions of Java, use Lombok to autogenerate the boilerplate methods.

Discussion

When Java was new, before there were good IDEs, developers had to write getters and setters by hand, or by copy-paste-change. Back then I did a study of one existing large codebase and found about a one-half of 1% failure rate. The setter stored the value in the wrong place or the getter retrieved the wrong value. Assuming random distribution, this meant that one getter call in a hundred gave the wrong answer! The application still worked, so I must assume those wrong answers didn't matter; presumably, the broken methods weren't actually called.

Now we have IDEs that can generate all the boilerplate methods such as setters/getters, equals, `toString()`, and so on. But you still have to remember to invoke these generators.

Java record type 16

The `record` type introduced in Java 16 provides a good solution for classes whose objects rarely or never need to be modifed. A `record` is a construct for data classes, a restricted form of class. You need only write the name of a data object and its fields, and the compiler will provide a constructor, getters, `hashCode()` and `equals()`, and `toString()`:

```
public record Person(String name, String emailAddress) { }
```

The provided constructor has the same signature as the record declaration. All fields are implicitly final, and the `record` provides getters but not setters. Because there can *never* be setters, the getters have just the name of the field; they do not follow the JavaBeans pattern of capitalizing the field name and prepending `get`, e.g., `getName()`. Immutable objects are important for reliable code (see Chapter 9). You can provide other members such as extra constructors, static fields, and static or instance methods. Records are final, cannot be abstract, and cannot declare additional instance fields. This is all in keeping with the fact that the state of the object is as declared in

the `record` header. Here I create a `Person` record and make an instance of it, all in JShell:

```
$ jshell
|  Welcome to JShell -- Version 16
|  For an introduction type: /help intro

jshell> record Person(String name, String email) {}

jshell> var p = new Person("Covington Roderick Smythe", "roddy@smythe.tld")
p ==> Person[name=Covington Roderick Smythe, email=roddy@smythe.tld]

jshell> p.name()
$3 ==> "Covington Roderick Smythe"

jshell>
```

One-line record definitions typically don't need to be in a source file all their own. To show a complete example, I baked the `Person` record into a new demo program `PersonRecordDemo`, shown in Example 8-2.

Example 8-2. main/src/main/java/record/PersonRecordDemo.java

```
public class PersonRecordDemo {

  /** This is all it takes to define a record -
   * compiler generates constructor, toString, equals/hashcode, etc.
   * Hint: after compiling this file, do:
   * javap 'PersonRecordDemo$Person'
   */
  public record Person(String name, String email) { }

  /**
   * Now we can use the record type as a regular data class.
   */
  public static void main(String[] args) {
    Person p = new Person("Covington Roderick Smythe", "roddy@smythe.tld");
    System.out.println(p);
  }
}
```

We can compile this file with `javac`, and then use `javap` to view the class's structure:

```
$ javac PersonRecordDemo.java
Note: PersonRecordDemo.java uses preview language features.
Note: Recompile with -Xlint:preview for details.
$ javap PersonRecordDemo'$'Person
Compiled from "PersonRecordDemo.java"
public final class PersonRecordDemo$Person extends java.lang.Record {
  public PersonRecordDemo$Person(java.lang.String, java.lang.String);
  public java.lang.String toString();
```

```
    public final int hashCode();
    public final boolean equals(java.lang.Object);
    public java.lang.String name();
    public java.lang.String email();
}
```

The *$* in the filename has to be escaped from the Unix shell. We see that the compiler has generated the constructor, `toString()`, `hashCode()` and `equals()`, and read-only accessors `name()` and `email()`.

Lombok

Project Lombok was an early solution to this problem, still used by many. Lombok processes classes containing for its annotations and rewrites the classes to have the chosen methods.

If you want to use Lombok, you will need to add the dependency `org.projectlombok:lombok:1.18.4` (or newer) to your build script. Or, if you are using an IDE, download the Lombok JAR file from the Project Lombok website and install it as per the instructions there. Then you can annotate your class with annotations like these:

```
@Setters @Getters
```

Presto! No more forgetting to generate these methods; Lombok will do the work for you.

Other annotations include the following:

```
@ToString
@EqualsAndHashCode
@AllArgsConstructor
```

For data classes, there is even `@Data`, which is a shortcut for `@ToString`, `@EqualsAndHashCode`, `@Getter` on all fields, `@Setter` on all nonfinal fields, and `@RequiredArgsConstructor`!

See Also

The original description of and rationale for the `record` mechanism is in JEP 359 at the OpenJDK website.

Lombok is implemented internally as an annotation processor, which runs at compile time. If you want more information on how this works, see Jacek Dubikowski's tutorial on annotation processors.

8.5 Providing Callbacks via Interfaces

Problem

You want to provide callbacks—that is, have unrelated classes call back into your code. For example, the observer class in the like-named pattern can be implemented by any class that implements the `Observer` interface.

Solution

One way to provide callbacks is to use a Java interface.

Discussion

An interface is a class-like entity that originally could contain only `abstract` methods and `final` fields. Later versions of Java had added `default` and `private` methods in interfaces. As we've seen, interfaces are used a lot in Java! In the standard API, the following are a few of the commonly used interfaces:

- `Runnable`, `Comparable`, and `Cloneable` (in `java.lang`).
- `List`, `Set`, `Map`, and `Enumeration`/`Iterator` (in the Collections API; as you'll see in Chapter 7).
- `ActionListener`, `WindowListener`, and others in the GUI layer.
- `Driver`, `Connection`, `Statement`, and `ResultSet` in JDBC (not officially an acronym, but widely understood as Java Database Connectivity); see the information on my website.
- The *remote interface*—the contract between the client and the server—is specified as an `Interface` (in some less-common Java Enterprise API network schemes: RMI [Remote Method Invocation], CORBA [Common Object Request Broker Interface], and EJB [Enterprise JavaBean]).

Subclass, Abstract Class, or Interface?

There is usually more than one way to solve a problem. Some problems can be solved by subclassing, by use of abstract classes, or by interfaces. The following general guidelines may help:

- A class can only extend one other class, but it can implement any number of interfaces; keep this in mind when deciding to use abstract classes or interfaces.
- *Use an abstract class* when you want to provide a template for a series of subclasses, all of which may inherit some of their functionality from the parent class

but are required to implement some of it themselves. (Any subclass of a geometric `Shape` class might have to provide a `computeArea()` method; because the top-level `Shape` class cannot do this, it would be abstract. This is implemented in Recipe 8.6.)

- *Use an interface with default methods* for many of the same things you'd have used an abstract class for prior to Java 8, which introduced default methods. Default methods can be added to an interface without breaking existing code, such as `forEach` and `of` methods being added to the `Collection` interface so that every `List` instance has a `forEach` method built in. One limitation is that interfaces cannot have nonpublic nondefault methods, as these are made public by default in interfaces.
- *Subclass* when you need to extend a class and add some functionality to it, whether the parent class is abstract or not. See the standard Java APIs and the examples in Recipes 2.9, 8.6, and 10.11
- *Subclass* when you are required to extend a given class. Some APIs such as *Servlets* use subclassing to ensure base functionality in classes that are dynamically loaded (see Recipe 17.1).
- *Define an interface* when there is no common parent class with the desired functionality and when you want only certain unrelated classes to have that functionality (see the `PowerSwitchable` interface later in this recipe). You should also choose this option if you know that you'll need (or think there is a chance you might later need) to be able to pass in unrelated classes for testing purposes. Using mock objects is a very common strategy in unit testing. Some say that interfaces should be your first choice at least as often as subclassing.
- *Use interfaces as markers* to indicate something about a class. Marker interfaces commonly have no abstract methods. The standard API, for example, uses `Serializable` as a marker interface to indicate permission to serialize objects of the implementing class. See Recipe 14.6 for information on serialization.

Suppose we are generating a building management system. To be energy efficient, we want to be able to remotely turn off (at night and on weekends) such things as room lights and computer monitors, which use a lot of energy. Assume we have some kind of remote control technology. It could be a commercial version of BSR's house-light control technology X10, it could be Bluetooth or 802.11—it doesn't matter. What matters is that we have to be very careful what we turn off. It would cause great ire if we turned off computer processors automatically—people often leave things running overnight. It would be a matter of public safety if we ever turned off the building's emergency lighting.[4]

4 Of course these lights wouldn't have remote power-off. But the computers might, for maintenance purposes.

So we've come up with the design shown in Figure 8-1.

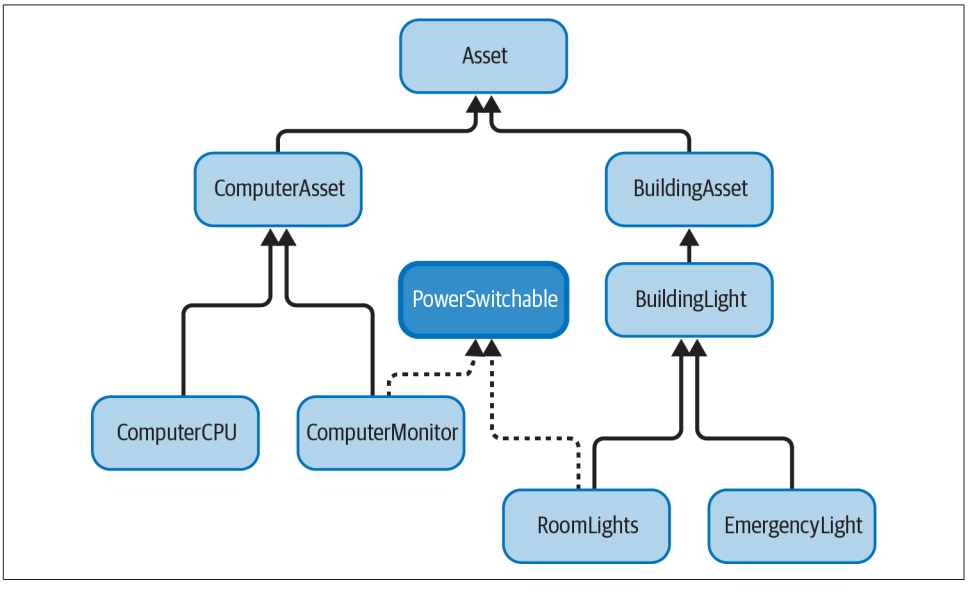


Figure 8-1. Classes for a building management system

The code for these data classes is not shown (it's pretty trivial), but it's in the *oo/interfaces* directory of the online source. The top-level classes (i.e., `BuildingLight` and `Asset`) are abstract classes. You can't instantiate them because they don't have any specific functionality. To ensure—both at compile time and at runtime—that we can never switch off the emergency lighting, we need only ensure that the class representing it, `EmergencyLight`, does not implement the `PowerSwitchable` interface.

Note that we can't very well use direct inheritance here. No common ancestor class includes both `ComputerMonitor` and `RoomLights` that doesn't also include `ComputerCPU` and `EmergencyLight`. Use interfaces to define functionality in unrelated classes.

How we use these is demonstrated by the `BuildingManagement` class; this class is not part of the hierarchy shown in Figure 8-1, but it *uses* a collection of `Asset` objects from that hierarchy.

Items that can't be switched off must nonetheless be in the database for various purposes (auditing, insurance, etc.). In the method that turns things off, the code is careful to check whether each object in the database is an instance of the `PowerSwitchable` interface. If so, the object is casted to `PowerSwitchable` so that its `powerDown()` method can be called. If not, the object is skipped, thus preventing any possibility of turning out the emergency lights or shutting off a machine that is busy running SETI@home, downloading a big MP3 playlist, or performing system backups. The following code shows this set of classes in action:

```
public class BuildingManagement {

  List<Asset> things = new ArrayList<>();

  /** Scenario: goodNight() is called from a timer Thread at 2200, or when
   * we get the "shutdown" command from the security guard.
   */
  public void goodNight() {
    things.forEach(obj -> {
      if (obj instanceof PowerSwitchable switchable) // Java 14 way
        switchable.powerDown();
      });
  }

  public void goodNightFunctional() {
    things.stream().filter(obj -> obj instanceof PowerSwitchable)
      .forEach(obj -> ((PowerSwitchable)obj).powerDown());
  }

  // goodMorning() would be similar, but call each one's powerUp().

  /** Add a Asset to this building */
  public void add(Asset thing) {
    System.out.println("Adding " + thing);
    things.add(thing);
  }

  /** The main program */
  public static void main(String[] av) {
    BuildingManagement b1 = new BuildingManagement();
    b1.add(new RoomLights(101));  // control lights in room 101
    b1.add(new EmergencyLight(101));  // and emerg. lights.
    // add the computer on desk#4 in room 101
    b1.add(new ComputerCPU(10104));
    // and its monitor
    b1.add(new ComputerMonitor(10104));

    // time passes, and the sun sets...
    b1.goodNight();
  }
}
```

When you run this program, it shows all the items being added but only the `PowerSwitchable` ones being switched off:

```
> java oo.interfaces.BuildingManagement
Adding RoomLights@2dc77f32
Adding EmergencyLight@2e3b7f32
Adding ComputerCPU@2e637f32
Adding ComputerMonitor@2f1f7f32
Dousing lights in room 101
Dousing monitor at desk 10104
>
```

8.6 Polymorphism/Abstract Methods

Problem

You want each of a number of subclasses to provide its own version of one or more methods.

Solution

Make the method abstract in the parent class; this makes the compiler ensure that it is implemented in each subclass.

Discussion

A hypothetical drawing program uses a `Shape` subclass for anything that is drawn. `Shape` has an abstract method called `computeArea()` that computes the exact area of the given shape:

```
public abstract class Shape {
    protected int x, y;
    public abstract double computeArea( );
}
```

A `Rectangle` subclass, for example, has a `computeArea()` that multiplies width times height and returns the result:

```
public class Rectangle extends Shape {
    double width, height;
    public double computeArea( ) {
        return width * height;
    }
}
```

A `Circle` subclass returns πr^2:

```
public class Circle extends Shape {
    double radius;
    public double computeArea( ) {
        return Math.PI * radius * radius;
    }
}
```

This system has a high degree of generality. In the main program, we can iterate over a collection of `Shape` objects and—here's the real beauty—call `computeArea()` on any `Shape` subclass object without having to worry about what kind of shape it is. Java's polymorphic methods automatically call the correct `computeArea()` method in the class of which the object was originally constructed:

main/src/main/java/oo/shapes/ShapeDriver.java

```
/** Part of a main program using Shape objects */
public class ShapeDriver {

  Collection<Shape> allShapes;  // created in a Constructor, not shown

  /** Iterate over all the Shapes, getting their areas;
   * this cannot use the Java 8 Collection.forEach because the
   * variable total would have to be final, which would defeat the purpose :-)
   */
  public double totalAreas() {
    double total = 0.0;
    for (Shape s : allShapes) {
      total += s.computeArea();
    }
    return total;
  }
```

Polymorphism is a great boon for software maintenance: if a new subclass is added, the code in the main program does not change. Further, all the code that is specific to, say, polygon handling, is all in one place: in the source file for the `Polygon` class. This is a big improvement over older languages, where type fields in a structure were used with case or `switch` statements scattered all across the software. Java makes software more reliable and maintainable with the use of polymorphism.

8.7 Improving Interfaces with Default, Static, and Private Methods

Problem

You want to use the recently added features in interface types.

Solution

Use default, static, and/or private methods.

Discussion

When Java was young, interfaces could only have abstract methods. Over the years they have gained several other abilities.

Default methods

Default methods were added in Java 8, and they allow for the addition of new methods in existing interfaces without breaking all implementations. This has been used in several interfaces, including some popular ones in the Collections Framework (see

Recipe 7.4). For example, as of Java 8, the `Iterable` interface has a `forEach` method that was not present early on; because this is a `default` method, all existing `List` and other `Collection` implementations gained the `forEach` method, without needing any changes! This is a prime example of how the Java core team keeps evolving the language without breaking backward compatibility. A simple implementation of this inside the JDK source `Iterable.java` would have been:

```
public interface Iterable {
        // ...
        default void forEach(Consumer c) {
                for (int i = 0; i < length(); i++) {
                        c.accept(get(i));
                }
        }
}
```

Since `Iterables` are in fact iterable, the actual code in the JDK is even simpler—the loop is simply:

```
for (T t : this) {
        c.accept(t);
}
```

Also note that, since all interface methods are now public, it is considered redundant to put the `public` modifier on any interface method, although omitting it is not yet universal.

Static and private methods in interfaces

Java 8 also allows interfaces to have static methods. These also must have a body, must be called by the interface's name, and are also implicitly public.

In Java 9, interfaces further gained the ability to have private methods. These are meant to be used from within `default` methods, and of course cannot be called from outside the interface.

Example 8-3 shows some examples of these changes.

Example 8-3. main/src/main/java/structure/InterfaceAdditions.java

```
public class InterfaceAdditions implements Demo {
  void main() { // Instance main in a named class
    // Invoke public interface method in implementing class
    System.out.println(getStatus());
    // Call static interface method
    System.out.println(Demo.getPresent());
    // Following cannot compile, private method is
    // only visible in the interface, not impl classes.
    // var s = innerStatus();
  }
```

```
}

interface Demo {

  default String getStatus() { return innerStatus(); }

  static String getPresent() { return "Freebie"; }

  private String innerStatus() { return "OK"; }
}

class InterfaceChangesII implements Demo {
  // Forllowing is a valid override, but must be public
  public String getStatus() { return "Unknown"; }
}
```

Languages that support multiple inheritance of classes, such as C++, have to deal with the issue of conflicting method names inherited from different classes, the so-called diamond inheritance problem, named as such because the inheritance tree forms a diamond shape. If there are conflicting methods in Java interfaces, the compiler will detect them, and you'd use *`interfacename`*`.super.methodName()` to disambiguate. There's an example in the online code in *main/src/main/java/lang/InterfaceConflicts.java*.

8.8 Using Typesafe Enumerations

Problem

You need to manage a small list of discrete values within a program.

Solution

Use the Java `enum` mechanism.

Discussion

To enumerate means to list all the values. You often know that a small list of possible values is all that's wanted in a variable, such as the months of the year, the suits or ranks in a deck of cards, or the primary and secondary colors. The C programming language provided an `enum` keyword:

```
enum { BLACK, RED, ORANGE} color;
```

Java was criticized in its early years for its lack of enumerations, which many developers have wished for. Many have had to develop custom classes to implement the *typesafe enumeration pattern*.

But C enumerations are not typesafe; they simply define constants that can be used in any integer context. For example, this code compiles without warning, even on older versions of GCC (those before 6.1.0) with `-Wall` (all warnings), whereas a C++ compiler catches the error:[5]

```
enum { BLACK, RED, ORANGE} color;
enum { READ, UNREAD } state;

/*ARGSUSED*/
int main(int argc, char *argv[]) {
        color = RED;
        color = READ; // In C this will compile, give bad results
        return 0;
}
```

To replicate this mistake in Java, one needs only to define a series of `final int` values; it will still not be *typesafe*. By typesafe I mean that you cannot accidentally use values other than those defined for the given enumeration. The definitive statement on the typesafe enumeration pattern is probably the version defined in item 21 of Joshua Bloch's book *Effective Java* (Addison-Wesley). All modern Java versions include enumerations; it is no longer necessary to use the code from Bloch's book. Bloch was one of the authors of the typesafe enumeration specification (`enum` keyword), so you can be sure that Java now does a good job of implementing his pattern. These `enum`s are implemented as classes, subclassed (transparently, by the compiler) from the class `java.lang.Enum`. Unlike C, and unlike a series of `final int`s, Java typesafe enumerations have the following qualities:

- They are printable (they print as the name, not as an underlying `int` implementation).
- They are almost as fast as `int` constants, but the code is more readable.
- They can be easily iterated over.
- They use a separate namespace for each `enum` type, which means you don't have to prefix each with some sort of constant name, like `ACCOUNT_SAVINGS`, `ACCOUNT_CHECKING`, etc.

`Enum` constants are not compiled into clients, giving you the freedom to reorder the constants within your `enum` without recompiling the client classes. That does not mean you should do this, however; think about the case where objects that use the enums have been persisted, and the person designing the database mapping used the numeric values of the `enum`s. Bad idea to reorder then!

5 For Java folks not that familiar with C/C++, C is the older, non-OO language; C++ is an OO derivative of C; and Java is in part a portable, more strongly typesafe derivative of C++.

Additionally, an `enum` type is a class, so it can, for example, implement arbitrary interfaces; and you can add constructors, fields, and methods to an `enum` class.

Compared to Bloch's Typesafe Enum pattern in the book:

- Java `enum`s are simpler to use and more readable (those in the book require a lot of methods, making them cumbersome to write).
- `Enum`s can be used in `switch` statements.

So there are many benefits and few pitfalls.

The `enum` keyword is at the same level as the keyword `class` in declarations. That is, an `enum` may be declared in its own file with public or default access. It may also be declared inside classes, much like nested or inner classes (see Recipe 8.3). *Media.java*, shown in Example 8-4, is a code sample showing the definition of a typesafe `enum`.

Example 8-4. structure/Media.java

```
public enum Media {
    BOOK, MUSIC_CD, MUSIC_VINYL, MOVIE_VHS, MOVIE_DVD;
}
```

Notice that an `enum` class *is* a class; see what `javap` thinks of the `Media` class:

```
    C:&gt; javap Media
    Compiled from "Media.java"
    public class Media extends java.lang.Enum{
        public static final Media BOOK;
        public static final Media MUSIC_CD;
        public static final Media MUSIC_VINYL;
        public static final Media MOVIE_VHS;
        public static final Media MOVIE_DVD;
        public static final Media[] values( );
        public static Media valueOf(java.lang.String);
        public Media(java.lang.String, int);
        public int compareTo(java.lang.Enum);
        public int compareTo(java.lang.Object);
        static {};
    }
```

Product.java, shown in Example 8-5, is a code sample that uses the `Media` enum.

Example 8-5. main/src/main/java/structure/Product.java

```
public class Product {
  String title;
  String artist;
  Media  media;
```

```
  public Product(String artist, String title, Media media) {
    this.title = title;
    this.artist = artist;
    this.media = media;
  }

  @Override
  public String toString() {
    switch (media) {
    case BOOK:
      return title + " is a book";
    case MUSIC_CD:
      return title + " is a CD";
    case MUSIC_VINYL:
      return title + " is a relic of the age of vinyl";
    case MOVIE_VHS:
      return title + " is on old video tape";
    case MOVIE_DVD:
      return title + " is on DVD";
    default:
      return title + ": Unknown media " + media;
    }
  }
}
```

In Example 8-6, `MediaFancy` shows how operations (methods) can be added to enumerations; the `toString()` method is overridden for the `Book` value of this enum.

Example 8-6. main/src/main/java/structure/MediaFancy.java

```
/** An example of an enum with method overriding */
public enum MediaFancy {
  /** The enum constant for a book, with a method override */
  BOOK {
    public String toString() { return "Book"; }
  },
  /** The enum constant for a Music CD */
  MUSIC_CD,
  /** ... */
  MUSIC_VINYL,
  MOVIE_VHS,
  MOVIE_DVD;

  /** It is generally discouraged to have a main() in an enum;
   * please forgive this tiny demo class for doing so.
   */
  public static void main(String[] args) {
    MediaFancy[] data =  { BOOK, MOVIE_DVD, MUSIC_VINYL };
    for (MediaFancy mf : data) {
      System.out.println(mf);
    }
```

```
  }
}
```

Running the `MediaFancy` program produces this output:

```
Book
MOVIE_DVD
MUSIC_VINYL
```

That is, the `Book` values print in a user-friendly way compared to the default way the other values print. In real life you'd want to extend this to all the values in the `enum`.

Finally, `EnumList`, in Example 8-7, shows how to list all the possible values that a given `enum` can take on; simply iterate over the array returned by the enumeration class's inherited `values()` method.

Example 8-7. structure/EnumList.java

```
public class EnumList {
  enum State {
    ON, OFF, UNKNOWN
  }
  public static void main(String[] args) {
    for (State i : State.values()) {
      System.out.println(i);
    }
  }
}
```

The output of the `EnumList` program is this, of course:

```
ON
OFF
UNKNOWN
```

8.9 Using Type Pattern Matching

Problem

You want to simplify code that processes multiple types of data.

Solution

Use type pattern matching.

Discussion

The concept of *type pattern matching* is usually referred to as just *pattern matching*. Unfortunately, at a glance this could be confused with regular expression pattern

matching, which is covered throughout Chapter 4. Type pattern matching applies to classes and class-like things, and starting with Java 21 (previously in preview) it can be used in `switch` statements to greatly simplify code.

Imagine a simple calculator program, in which the parser has constructed objects whose types represent operations such as `Add`, `Subtract`, and so on, all of which extend class `Op`. A second pass will evaluate and print each operation. Prior to type pattern matching in `switch` (introduced in Java 21) the code might look like Example 8-8.

Example 8-8. main/src/main/java/lang/PatternMatchingString.java (old way)

```
System.out.println("Prior to type pattern matching:");
for (Op op : ops) {
  if (op instanceof Add) {
    System.out.println(((Add)op).left + ((Add)op).right);
  }
  else if (op instanceof Subtract) {
    System.out.println(((Subtract)op).left - ((Subtract)op).right);
  }
  else if (op instanceof Multiply) {
    System.out.println(((Multiply)op).left * ((Multiply)op).right);
  }
  else if (op instanceof Divide) {
    System.out.println(((Divide)op).left / ((Divide)op).right);
  }
  else System.out.println("Dunno about " + op);
}
```

This works, but it is fairly verbose. With Java 21, the code could be written more concisely, as in Example 8-9.

Example 8-9. main/src/main/java/lang/PatternMatchingString.java (new way)

```
System.out.println("Using type pattern matching:");
for (Op op : ops) {
  System.out.println(
    switch(op) {
    case null       -> "Null is not a valid Op!";
    case Add      a -> a.left + a.right;
    case Subtract s -> s.left - s.right;
    case Multiply m -> m.left * m.right;
    case Divide   d -> d.left / d.right;
    }
  );
}
```

For example, the code `case Add a -> a.left + a.right` says that if `op` is of type `Add`, then implicitly cast it to type `Add`, to be able to refer to its fields without an explicit cast.

19 Pattern matching can also be used to deconstruct or destructure `records`. Here again we put the variables after the type, but in this form the variables can be a `record` type with a list of fields to extract. This is shown, both using `instanceof` and using a `switch`, in Example 8-10.

Example 8-10. main/src/main/java/lang/PatternRecordDestruction.java

```
record Graduate(String name, String degree, int year){}

void main() {

  // with instanceof:
  for (Object o : grads) {
    if (o instanceof Graduate(var nm, var deg, var year)) {
      System.out.printf("%s graduated %d with a %s\n", nm, year, deg);
    }
  }

  // pattern match in switch:
  for (Object o : grads) {
    switch(o) {
      case Graduate(var nm, var deg, var year) ->
        System.out.printf("%s graduated %d with a %s\n", nm, year, deg);
      default -> {}
    }
  }
}
```

See Also

For a few nuances and for more on the rationale behind pattern matching and unpacking, see JEP 441.

8.10 Avoiding NPEs with "Optional"

Problem

You worry about null references causing a `NullPointerException` (NPE) in your code.

Solution

Use `java.util.Optional`.

Discussion

The computer scientist and professor who invented the notion of null pointers, a key early contributor to our discipline, has described the null reference as "my billion-dollar mistake". However, use of `null` is not going away anytime soon.

What we can do is make clear that we worry about null pointers in certain contexts. For this purpose, Java 8 introduced the class `java.util.Optional`. The `Optional` is an object wrapper around a possibly null object reference. The notion of an `Optional` wrapper has a long history; a similar construct is found in several functional programming languages, including Scala and Haskell, and in LLVM's ADT.

Java `Optionals` can be created with one of these methods:

`Optional.empty()`
: Returns an empty optional

`Optional.of(T obj)`
: Returns a nonempty optional containing the given value

`Optional.ofNullable(T obj)`
: Returns either an empty optional or one containing the given value

The basic operation of this class is to behave in one of two ways, depending on whether it is full or empty. Optional objects are immutable, so they cannot transition from one state to the other.

The simplest use is to invoke `isEmpty()` or its opposite, `isPresent()`, and use program logic to behave differently. This is not much different from using an `if` statement to check for null, but it puts the choice in front of you, making it less likely that you'll forget to check:

```
jshell> Optional<String> opt = Optional.of("What a day!");
opt ==> Optional[What a day!]

jshell> if (opt.isPresent()) {
   ...>     System.out.println("Value is " + opt.get());
   ...> } else {
   ...>     System.out.println("Value is not present.");
   ...> }
Value is What a day!
```

A better form would use the `orElse` method:

```
jshell> System.out.println("Value is " + opt.orElse("not present"));
Value is What a day!

jshell> opt = Optional.empty();
opt ==> Optional.empty
```

```
jshell> System.out.println("Value is " + opt.orElse("not present"));
Value is not present
```

One use case is that of passing values into methods. The object can be wrapped in an `Optional` either before it is passed to a method or after; the latter is useful when migrating from code that didn't use `Optional` from the start. The `Item` demo in Example 8-11 might represent part of a shipment-tracking program, a lending library manager, or anything else that has time-related data which might be missing.

Example 8-11. main/src/main/java/oo/OptionalDemo.java

```
    List.of(
      new Item("Item 1", LocalDate.now().plusDays(7)),
      new Item("Item 2")).
        forEach(System.out::println);
  static class Item {
    String name;
    Optional<LocalDate> dueDate;
    Item(String name) {
      this(name, null);
    }
    Item(String name, LocalDate dueDate) {
      this.name = name;
      this.dueDate = Optional.ofNullable(dueDate);
    }

    public String toString() {
      return "%s %s".formatted(name,
        dueDate.isPresent() ?
          "Item is due on " + dueDate.get() :
          "Sorry, do not know when item is due");
    }
  }
```

There are methods that throw exceptions, that return null, and so on. There are also methods for interacting with the `Streams` mechanism (see Recipe 9.3). These methods provide a major simplification, in that we don't have to check for null until we need to see the final result (for example, with Stream terminal methods returning `Optional`). A full list of `Optional`'s methods is at the start of the Javadoc page.

8.11 Controlling Subclassing with Sealed Types 17

Problem

You want to limit subclassing without making your types `final`.

Solution

Use sealed classes (or sealed interfaces).

Discussion

Until sealed classes were added, Java only allowed types to be `final` or to allow unlimited subclassing. With sealed classes, you can now provide an exhaustive list of subtypes; no other subtypes will be allowed.

Sealed types apply to classes and interfaces, but not to records, enums, or annotations.

The syntax is:

```
[public] sealed Type TypeName permits SubType1, SubType2, ...;
```

where the word `Type` can be either a class or an interface. Only the listed `SubType` classes will be allowed to directly extend `TypeName`. Subtypes of a sealed class must be declared either `sealed`, `non-sealed` (yes, that is a dash inside a Java language keyword), or `final`. Use:

- `final` if you want no further subclassing
- `sealed` with a `permits` list if you want controlled further subclassing
- `non-sealed` if you want uncontrolled subclassing from this particular type

A real benefit of this is that, when these types are used in a `switch` statement, the compiler can be sure that all possible types have been covered, obviating the need for a `default` case.

The code in Example 8-12 shows some examples.

Example 8-12. main/src/main/java/sealedclasses/simple_example.java

```
sealed class C1 permits C2, C3 {}  // Only 2 direct subclasses
final class C2 extends C1 {}    // cannot be subclassed
non-sealed class C3 extends C1 {}  // Valid subclass, wide open for subclassing!
class C4 extends C3 {}        // Subclass of C3
class SomeRandomUserClass extends C3 {} // Another subclass of C3
```

Figure 8-2 illustrates the possible subclasses from class `C1`.

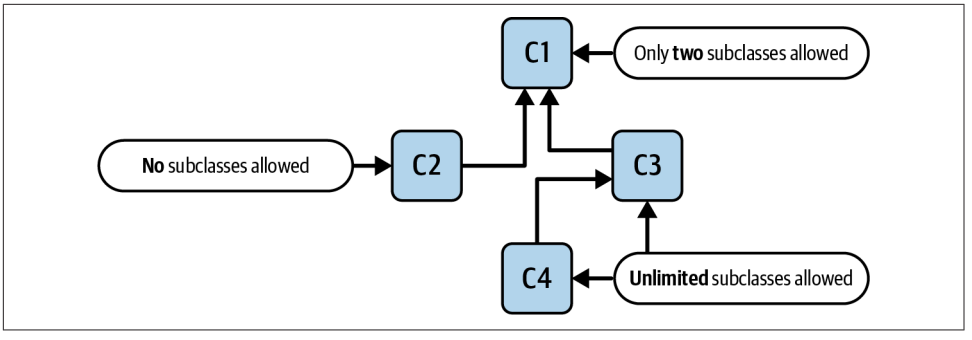


Figure 8-2. Sealed types inheritance

Sealed interfaces work the same way. A sealed interface only permits the named subinterfaces. The subinterfaces can be sealed or non-sealed, but not final, of course. This is illustrated by Example 8-13.

Example 8-13. main/src/main/java/sealedclasses/sealed_interfaces.java

```
void main() {
  System.out.println("Nothing to see here, folks.");
}

sealed interface A1 permits A2, A3, A4 {
  void process();
}

non-sealed interface A2 extends A1 {}

interface B1 extends A2 {}    // Allowed because A2 not sealed

non-sealed interface A3 extends A1 {}

sealed interface A4 extends A1 permits B2 {}

non-sealed interface B2 extends A4 {}
```

8.12 Enforcing the Singleton Pattern

Problem

You want to be sure there is only one instance of your class in a given Java Virtual Machine, or at least within your application.

Solution

There are several methods for making your class enforce the Singleton pattern:

- Using `enum` implementation
- Having only a private constructor (or multiple) and a `getInstance()` method
- Using a framework such as Spring or CDI (Recipe 8.14) configured to give Singleton-style instantiation of plain classes

Discussion

It is often useful to ensure that only one instance of a class gets created, usually to funnel all requests for some resource through a single point. An example of a Singleton from the standard API is `java.lang.Runtime`: you cannot create instances of `Runtime`; you simply ask for a reference by calling the static method `Runtime.getRuntime()`. Singleton is also an example of a design pattern that can be easily implemented. In all forms, the purpose of the Singleton implementation is to provide an instance in which certain methods can run, typically to control access to some resource.

The easiest implementation uses a Java `enum` to provide Singleton-ness. The `enum` mechanism already guarantees that only one instance of each enum constant will exist in a given JVM context, so this technique piggy-backs on that, as shown in Example 8-14.

Example 8-14. main/src/main/java/oo/EnumSingleton.java

```
public enum EnumSingleton {

    INSTANCE;

    // instance methods protected by singleton-ness would be here...

    /** A simple demo method */
    public String demoMethod() {
        return "demo";
    }
}
```

Using it is simple:

```
// Demonstrate the enum method:
EnumSingleton.INSTANCE.demoMethod();
```

The next easiest implementation consists of a private constructor and a field to hold its result, as well as a static accessor method with a name like `getInstance()`.

The private field can be assigned from within a static initializer block or, more simply, by using an initializer. The `instance()` method (which should be public and must be static) then simply returns this instance, as shown in Example 8-15.

Example 8-15. main/src/main/java/oo/Singleton.java

```
public class Singleton {

    /**
     * Static initializer is run before class is available to code, avoiding
     * broken antipattern of lazy initialization in instance method.
     * For more complicated construction, could use static block initializer.
     */
    private static Singleton instance = new Singleton();

    /** A private Constructor prevents any other class from instantiating. */
    private Singleton() {
        // nothing to do this time
    }

    /** Static 'instance' method to obtain the one-and-only instance */
    public static Singleton instance() {
        return instance;
    }

    // other instance methods protected by singleton-ness would be here...

    /** A simple demo method */
    public String demoMethod() {
        return "demo";
    }
}
```

Note that the method of using *lazy evaluation* in the `getInstance()` method (as demonstrated in the book *Design Patterns*) is not necessary in Java because Java already uses *lazy loading*. Your `Singleton` class will probably not get loaded until its `getInstance()` is called, so there is no point in trying to defer the `Singleton` construction until it's needed by having `getInstance()` test the `singleton` variable for null and creating the `Singleton` there.

Using this class is equally simple: just get the instance reference, and invoke methods on it:

```
// Demonstrate the codeBased method:
Singleton.instance().demoMethod();
```

Some commentators believe that a code-based Singleton should also provide a `public final clone()` method that just throws an exception, in order to avoid subclasses that cheat and `clone()` the Singleton. However, it is clear that a class with only a private constructor cannot be subclassed, so this paranoia does not appear to be necessary.

See Also

The `Collections` class in java.util has methods `singletonList()`, `singletonMap()`, and `singletonSet()`, which give out an immutable `List`, `Map`, or `Set`, respectively, containing only the one object that is passed to the method. This does not, of course, convert the object into a Singleton in the sense of preventing that object from being cloned or other instances from being constructed.

See discussions in the original *Design Patterns* book and the book *Effective Java*.

8.13 Roll Your Own Exceptions

Problem

You'd like to use an application-specific exception class or two.

Solution

Go ahead and subclass `Exception` or `RuntimeException`.

Discussion

In theory, you could subclass `Throwable` directly, but that's considered rude. You normally subclass `Exception` (if you want a checked exception) or `RuntimeException` (if you want an unchecked exception). Checked exceptions are those that an application developer is required to either catch or delegate upward by listing them in the `throws` clause of the invoking method.

When subclassing either of these, it is customary to provide at least these constructors:

- A no-argument constructor
- A one-string argument constructor
- A two-argument constructor—a string message and a `Throwable` cause

The cause will appear if the code receiving the exception performs a stack trace operation on it, with the prefix `Root Cause is` or similar. Example 8-16 shows these three constructors for an application-defined exception, `ChessMoveException`.

Example 8-16. main/src/main/java/oo/ChessMoveException.java

```
/** A ChessMoveException is thrown  when the user makes an illegal move. */
public class ChessMoveException extends Exception {

  private static final long serialVersionUID = 8029117369881790791;
```

```
  public ChessMoveException () {
    super();
  }

  public ChessMoveException (String msg) {
    super(msg);
  }

  public ChessMoveException(String msg, Exception cause) {
    super(msg, cause);
  }
}
```

The Javadoc documentation for `Exception` lists a large number of subclasses; you might look there first to see if there is one you can use, before creating your own.

8.14 Using Dependency Injection

Problem

You want to avoid excessive coupling between classes, and you want to avoid excessive code dedicated to object creation/lookup.

Solution

Use a dependency injection (DI) framework.

Discussion

A DI framework allows you to have objects passed into your code instead of making you either create them explicitly (which ties your code to the implementing class name, since you're calling the constructor) or looking for them (which requires use of a possibly cumbersome lookup API, such as JNDI, the Java Naming and Directory Interface).

When using a DI framework, it's common to provide a Java interface for each class that you are working with, which this reduces coupling; that is, it removes dependencies on particular implementation classes and allows easier substitution of implementations when testing or when needs change or more flexibility is needed.

Three of the best-known DI frameworks are the Spring Framework (Spring DI), the Java Enterprise Edition's Contexts and Dependency Injection (CDI) (recently migrated to Jakarta), and Google Guice.

This example is a simplification of a numerical processing application. It uses Spring DI because it's relatively easy to work with. We have a main program that starts up Spring and requests an implementation of an interface called `Processor`. `Processor` does some numeric processing, which might be complex enough to use the Vector API described in Recipe 5.11. The `Processor` implementation is called `Process`, and because it is the top of the hierarchy, we'd have to either give its class name hardcoded in the main program, or provide an `@Configuration`-annotated class, which externalizes the dependency of the main program on the `Processor` implementation.

The `Processor` also needs a `Reporter` implementation to report its output; this might allow you to choose between a console reporter, a Swing reporter, a reporter that stores the results in a spreadsheet or database, etc. Given that the `Processor` has been obtained from the Spring factory (`ApplicationContext`), we can use Spring DI's `@AutoWire` annotation to inject the `Reporter` implementation into it:

```
SpringMain - main program
`-- Processor - obtained from Context, via BeanConfig
    `-- Reporter - obtained automatically via @AutoWired
```

The `Processor` and `Reporter` interfaces are trivial; each has one method with a name similar to the interface name (see Example 8-17).

Example 8-17. Dependency interfaces

```
include: main/src/main/java/di0/Process.java
include: main/src/main/java/di0/Reporter.java
```

The main program is shown here:

main/src/main/java/di0/SpringMain.java

```
public class SpringMain {
  public static void main(String[] args) {
      ApplicationContext ctx =
         new AnnotationConfigApplicationContext("di0");

      Processor processor = ctx.getBean(Processor.class);

      processor.process(2,3);
  }
}
```

This class has to:

1. Set up the Spring context, which provides the DI framework.
2. Get the `Processor` from the context; it will already have the `Reporter` plugged into it by the DI framework.

3. Ask the `Processor` to process some data.

Furthermore, we don't depend on particular implementations of the interface.

How does Spring know to inject, or provide, a `Reporter` into the `Processor`? And what if there are multiple implementations of the `Reporter` interface? We have to tell it these things, which we'll do here with annotations:

```
@Configuration
public class SpringBeanConfig {
    @Bean
    public Processor processor() {
        return new Process();
    }
}
```

While Spring originally provided its own annotations, it will also accept the Java standard `@jakarta.annotation.Resource` annotation for injection and `@jakarta.inject.Named` to specify the injectee.

The `Processor` has the `Reporter` injected into it by the DI framework, just by asking for it with the `@AutoWired` annotation:

```
@Named("processor")
public class Process implements Processor {

        @Autowired
        private Reporter reporter;

        public void process(int x, int y) {
                var result = 2 * x + 3 * y;
                reporter.report(result);
        }
}
```

If you are familiar with Java design patterns, you might recognize this as a very simplified Model-View-Controller (MVC) application. A more explicit MVC version is in the *main/src/main/java/di* directory.

See Also

Spring DI, Java EE CDI, and Guice all provide powerful dependency injection. Spring's framework is more widely used; Java EE's framework has the same power and is built into every EE container. All three can be used alone or in a web application, with minor variations. The overall Spring Framework provides special support for web apps, and CDI is already set up in Jakarta EE containers. There are many books on Spring. One book specifically treats Weld: *JBoss Weld CDI for Java Platform* by Ken Finnigan (O'Reilly).

8.15 Combining Java Features for Data-Oriented Programming

Problem

You want to make your applications more effective by combining such new features as `records`, enhanced `switch` with pattern matching, and sealed types.

Solution

Learn the techniques of the data-oriented programming (DOP) paradigm.

Discussion

Like functional programming (Chapter 9), DOP is not a replacement for object-oriented programming, but an adjunct to it. However, DOP does walk back a bit from one key tenet of OOP: the combination of state and processing. Instead, DOP advocates using Java's `record` type as largely behaviorless data objects. Some of the main points are:

Modeling data as data
: Use features like `record` (Recipe 8.4) and sealed classes (Recipe 8.11) for reliable data representation. This is intended to improve code readability and make it easier to reason about how data is used.

Separation of concerns
: Treat data validation, data transport/storage, and data processing as distinct.

Immutable data
: Make data objects immutable to increase reliability and make it easier to reason about data.

Pattern matching
: Simplify data handling by using pattern matching in record unpacking and switches (with `instanceof`).

A simplified example is in Example 8-18. This provides a sample slice of an application used to bill customers for their orders, perhaps weekly.

Example 8-18. main/src/main/java/oo/DataOrientedDemo.java

```
sealed interface Transaction permits Order, Refund {}  ❶

record Order(String product, int quantity,             ❷
  double price) implements Transaction {}
```

```
record Refund(String reason,
  double amount) implements Transaction {}

record Customer(String name, String address,          ❸
  String State, List<Transaction> orders) {}

void process(List<Customer> customers) {
  for (Customer c : customers) {
    double balance = 0;                  ❹
    for (Transaction tx: c.orders) {
      switch (tx) {
      case Order(String p, int q, double price) -> {  ❺
        System.out.printf("Invoice %s for %d %d for $%7.2f\n",
          c.name(), q, p, price);
        balance += price;
      }
      case Refund(String reason, double amount) -> {
        System.out.printf("Credit %s $%7.2f due to %s\n",
          c.name(), amount, reason);
        balance -= amount;
      }
      //                        ❻
      }
    }
    System.out.printf("Net balance for %s is %f", c.name(), balance); ❼
  }
}

void main() {
  List<Transaction> txlist = List.of(new Order("Widgets", 10, 200.00),
    new Order("Whatzits", 5, 125.00),          ❽
    new Refund("Lost shipment #456", 175.00));
    var cust =
      new Customer("Whizzy Systems Inc", "123 Erewhon St", "Confusion", txlist);
  process(List.of(cust));
}
```

❶ Using a `sealed` interface lets the compiler be sure we handle all cases.

❷ Records can implement interfaces; we implement `Transaction` here.

❸ `Customer` is also a record, making all our data types immutable and thus thread-safe.

❹ Set up for item 5.

❺ Processing is done here, not in methods of `Order`, `Refund`, or `Customer`.

❻ No `default` case; see item 1.

❼ Printing out the balance concludes the processing started in item 5.

❽ This dummy data stands in for code that would receive *and validate* the `Transaction` and `Customer` objects, making our data "safe" in the rest of the code.

See Also

Brian Goetz's InfoQ article on DOP was part of the Java team's push to promote the use of DOP. Another Java team member, Nicolai Parlog, wrote this six-part series on DOP.

CHAPTER 9

Functional Programming Techniques: Functional Interfaces, Streams, and Parallel Collections

9.0 Introduction

Java is an object-oriented (OO) language. You know what that is. Functional programming (FP) has been attracting attention lately. There may not be quite as many definitions of FP as there are FP languages, but it's close. Wikipedia's definition of functional programming is as follows (viewed December 2013):

> a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data. Functional programming emphasizes functions that produce results that depend only on their inputs and not on the program state—i.e. pure mathematical functions. It is a declarative programming paradigm, which means programming is done with expressions. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) both times. Eliminating side effects, i.e. changes in state that don't depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

How can we benefit from the FP paradigm? One way would be to switch to using an FP language; some of the leading languages are Haskell,[1] Idris, OCaml, Erlang, Julia, and the LISP family. But most of those would require walking away from the Java

1 Haskell was used to write a fairly complete Twitter clone in a few hundred lines; see simple-twitter on GitHub.

ecosystem. You could consider using Scala or Clojure, JVM-based languages that provide functional programming support in the context of an OO language.

But this is the *Java Cookbook*, so as you can imagine, we're going to try to get as many benefits of functional programming as we can while remaining in the Java language. Some features of FP include the following:

- Pure functions that have no side effects and whose results depend only on their inputs and not on mutable state elsewhere in the program
- First-class functions (e.g., functions as data)
- Immutable data
- Extensive use of recursion and lazy evaluation

Pure functions are completely self-contained; their operation depends only on the input parameters and internal logic, not on any variable state in other parts of the program—indeed, there are no global variables, only global *constants*. Although this can be hard to accept for those schooled in imperative languages like Java, it does make it much easier to test and ensure program correctness! It means that, no matter what else is going on in the program (even with multiple threads), a method call like `computeValue(27)` will always, unconditionally, return the same value every time (with exceptions, of course, for things like the current time, random seeds, etc., which are global state).

We'll use the terms *function* and *method* interchangeably in this chapter, although this is not strictly correct. FP people use the term *function* in the mathematical function sense, whereas in Java *methods* just means some code you can call (a Java method call is also referred to as a *message* being *sent* to an object, in the OO view of things).

Functions as data means that you can create *an object that is a function*, pass it into another function, write a function that returns another function, and so on—with no special syntax, because, well, functions *are* data.

One of Java's approaches to FP is the definition of functional interfaces. A *functional interface* in Java is one that has only one abstract method, such as the widely used `Runnable`, whose only method is `run()`, or the common Swing action handler `ActionListener`, whose only method is `actionPerformed(ActionEvent)`. As we saw in Recipe 8.7, Java 8+ allows interfaces to have `default` methods, which are available for use in any class that `implements` the interface. Such methods cannot depend on instance state in a particular class because they would have no way of referring to it at compile time.

So a functional interface is more precisely defined as one that has a single nondefault method. You can do functional-style programming in Java if you use functional interfaces and if you restrict code in your methods to not depending on any nonfinal instance or class fields; using `default` methods is one way of achieving this. The first few recipes in this chapter discuss functional interfaces.

Another Java approach to functional-ness is the *lambda expression*. A lambda is an expression of a functional interface, and it can be used as data (i.e., assigned, returned, etc.). Just to give a couple of short examples for now:

```
ActionListener x = e -> System.out.println("You activated " + e.getSource());
```

```
public class RunnableLambda {

  private static ExecutorService threadPool =
    Executors.newSingleThreadExecutor();

  public static void main(String[] args) {
    threadPool.submit(() -> System.out.println("Hello from a thread"));
```

Immutable data is easy in theory: just have a class with only read accessors (`get` methods). The downside is it takes some getting used to. But it is worthwhile and it works. The standard `String` class, for example, is immutable: methods like `substring()` or `toUpperCase()` don't change the original string, but make up new string objects with the requested change. Yet strings are universally used, and useful. Enums are also implicitly immutable. Java 16 added a new kind of class-like object called a `record`; `record`s are implicitly immutable. The compiler generates `get`-like methods for the fields (along with a constructor and the common `Object` methods), but no `set` methods.

Java 8 also introduced `Stream` classes. A `Stream` is like a pipeline that you can feed into, fan out, collect down—like a cross between the Unix notion of pipelines and Google's distributed programming concept of MapReduce, as exemplified in Hadoop, but running in a single program in a single VM. `Stream`s can be sequential or parallel; the latter are designed to take advantage of the massive parallelism that is happening in hardware design (particularly servers, where 16-or-more-core processors are popular). We discuss `Stream`s in several recipes in this chapter.

If you're familiar with Unix pipes and filters, this equivalence will make sense to you; if not, you can skip it for now. The Unix command is this:

```
cat lines.txt | sort | uniq | wc -l
```

The Java `Stream`s equivalent is shown in Example 9-1. For small inputs, the Unix pipeline is faster; but for larger volumes, the Java one should be faster, especially when parallelized.

Example 9-1. main/src/main/java/functional/UnixPipesFiltersReplacement.java

```
long numberLines = Files.lines(Path.of(("lines.txt")))
  .sorted()
  .distinct()
  .count();
System.out.printf("lines.txt contains " + numberLines + " unique lines.");
```

Tied in with `Streams` is the notion of a `Spliterator`, a derivative (logically, not by inheritance) of the familiar `Iterator` but designed for use in parallel processing. Most users will not be expected to develop their own `Spliterator` and will likely not even call its methods directly very often, so we do not discuss it in detail.

See Also

For general information on functional programming, see the book *Functional Thinking* by Neal Ford (O'Reilly).

There is an entire book dedicated to lambda expressions and related tools, Richard Warburton's *Java 8 Lambdas* (O'Reilly).

9.1 Using Lambdas/Closures Instead of Inner Classes

Problem

You want to avoid all the typing that even the anonymous style of inner class requires.

Solution

Use Java's lambda expressions.

Discussion

The symbol lambda (λ) is the 11th letter of the Greek alphabet and thus as old as Western society. The Lambda calculus is older than our notions of computing. In this context, lambda expressions are small units of calculation that can be referred to. They are functions as data. In that sense, they are a lot like anonymous inner classes, though it's probably better to think of them as *anonymous methods*. They are essentially used to replace inner classes for a *functional interface*—that is, an interface with one abstract method (function) in it. A desktop example is the AWT `ActionListener` interface, widely used in GUI code, whose only method is this one:

```
public void actionPerformed(ActionEvent e);
```

Using lambdas is now the preferred method of writing for GUI action listeners. Here's a single example:

```
quitButton.addActionListener(e -> shutDownApplication(0));
```

Closures in computing are functions that capture (have access to) variables from their surrounding environment. This allows them to access and modify variables from outside their scope. Closures are not explicitly supported in Java, but can be simulated using lambdas with effectively final variables.

Because fewer developers write Swing GUI applications these days, let's start with an example that doesn't require GUI programming. Suppose we have a collection of camera model descriptor objects that has already been loaded from a database into memory, and we want to write a general-purpose API for searching them, for use by other parts of our application.

The first thought might be along the following lines:

```
public interface CameraInfo {
    public List<Camera> findByMake();
    public List<Camera> findByModel();
    ...
}
```

Perhaps you can already see the problem. You will also need to write `findByPrice()`, `findByMakeAndModel()`, `findByYearIntroduced()`, and so on as your application grows in complexity.

You could consider implementing a query by example method, where you pass in a `Camera` object and all its nonnull fields are used in the comparison. But then how would you implement finding cameras with interchangeable lenses *under $500*?[2]

So a better approach is probably to use a callback function to do the comparison. Then you can provide an anonymous inner class to do any kind of searching you need. You'd want to be able to write callback methods like this:

```
public boolean choose(Camera c) {
    return c.isIlc() && c.getPrice() < 500;
}
```

2 If you ever have to build this kind of service where the data is stored in a relational database using the Java Persistence API (JPA), you should check out the Spring Data or Jakarta Data frameworks. These allow you to define an `interface` with method names like `findCameraByInterchangeableTrueAndPriceLessThan(double price)` and have the framework implement these methods for you.

Accordingly, we'll build that into an interface:[3]

```
/** An Acceptor accepts some elements from a Collection */
public interface CameraAcceptor {
    boolean choose(Camera c);
}
```

Now the search application provides a method:

```
public List<Camera> search(CameraAcceptor acc);
```

which we can call with code like this:

```
results = searchApp.search(new CameraAcceptor() {
    public boolean choose(Camera c) {
        return c.isIlc() && c.getPrice() < 500;
    }
}
```

Or, if you are not comfortable with anonymous inner classes, you might have to type this:

```
class MyIlcPriceAcceptor implements CameraAcceptor {
    public boolean choose(Camera c) {
        return c.isIlc() && c.getPrice() < 500;
    }
}
CameraAcceptor myIlcPriceAcceptor = new MyIlcPriceAcceptor();
results = searchApp.search(myIlcPriceAcceptor);
```

That's really a great deal of typing just to get one method packaged up for sending into the search engine. Java's support for lambda expressions or closures was argued about for many years (literally) before the experts agreed on how to do it. And the result is staggeringly simple. One way to think of Java lambda expressions is that each one is just a method that implements a functional interface. With lambda expressions, you can rewrite the preceding code as just:

```
results = searchApp.search(c -> c.isIlc() && c.getPrice() < 500);
```

The arrow notation `->` indicates the code to execute. If it's a simple expression, as we have here, you can just write it as shown. If there is conditional logic or other statements, you have to use a block, as is usual in Java.

3 If you're just not that into cameras, the description "Interchangeable Lens Camera (ILC)" includes two categories of what you might find in a camera store: traditional DSLR (Digital Single Lens Reflex) cameras and the newer category of Mirrorless Cameras like the Nikon Z-30/Z50, Sony ILCE (formerly known as NEX), and the Canon EOS-M, all of which are smaller and lighter than the older DSLRs. Both types allow the use of different lenses, usually through a twist-lock mounting system that holds the lens in place on the camera body.

Here is the search example rewritten (more verbosely than needed) to show how it would look as a block:

```
results = searchApp.search(c -> {
    if (c.isIlc() && c.getPrice() < 500)
        return true;
    else
        return false;
});
```

The first `c` inside the parenthesis corresponds to `Camera c` in the explicitly implemented `choose()` method: you can omit the type because the compiler knows it! If there is more than one argument to the method, you must parenthesize them. Suppose we had a compare method that takes two cameras and returns a quantitative value (oh, and good luck trying to get two photographers to agree on *that* algorithm!):

```
double goodness = searchApp.compare((c1, c2) -> {
    // write some amazing code here
});
```

This notion of *lambdas* seems pretty potent, and it is! You will see much more of this in Java as the use of lambdas moves into the mainstream of computing.

Up to this point, we still have to write an interface for each type of method that we want to be able to lambda-ize. The next recipe shows some predefined interfaces that you can use to further simplify (or at least shorten) your code.

And, of course, there are many existing interfaces that are functional, such as the `ActionListener` interface from GUI applications. Interestingly, the IntelliJ IDE (see Recipe 1.4) automatically recognizes inner class definitions that are replaceable by lambdas and, when using *code folding* (the IDE feature that represents an entire method definition with a single line), displays the inner class as the corresponding lambda, but leaves it as an inner class in the code. Modern versions will also offer to replace the entire inner class definition with a lambda when you mouse over the class name in `new ActionListener…`. Figures 9-1 and 9-2 show a before-and-after picture of this code folding. Figure 9-3 shows the IDE offering to replace the inner class with a lambda.

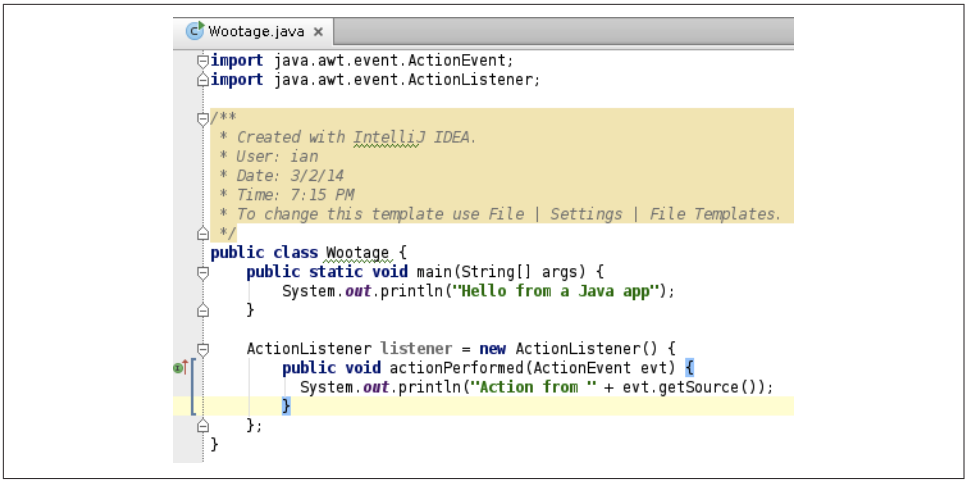


Figure 9-1. IntelliJ code unfolded

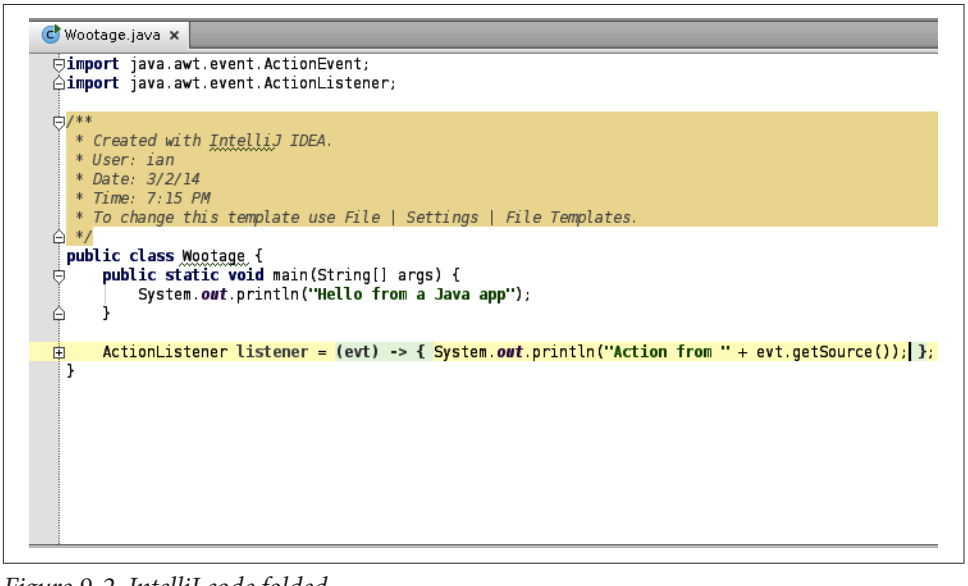


Figure 9-2. IntelliJ code folded

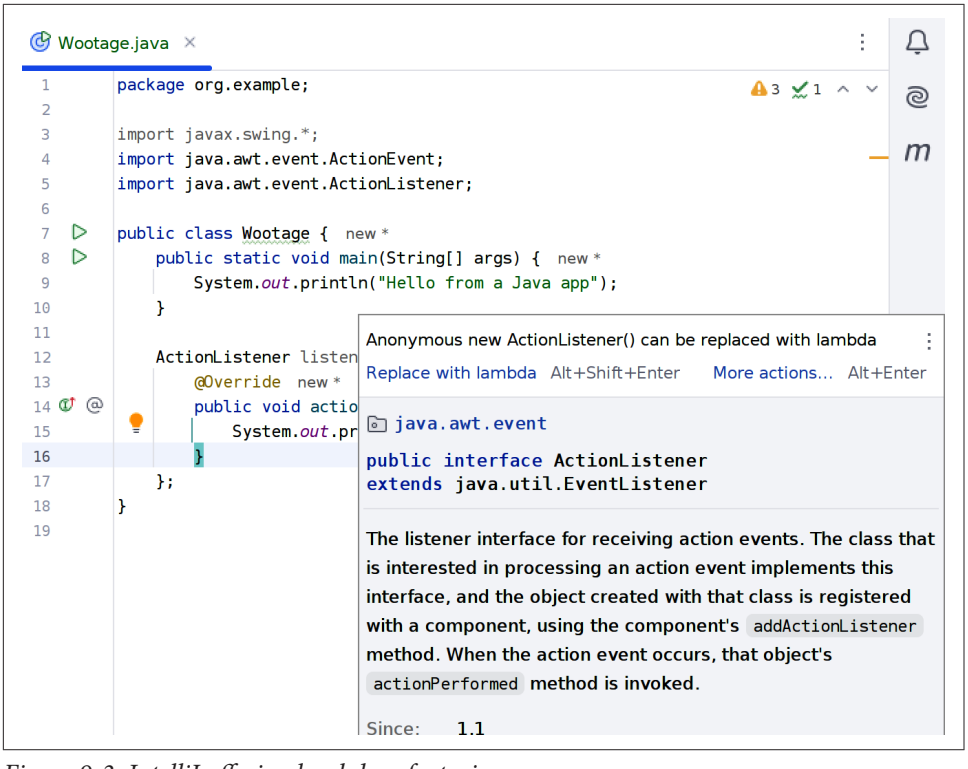

Wootage.java
package org.example;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class Wootage {
public static void main(String[] args) {
System.out.println("Hello from a Java app");
ActionListener listen
@Override
public void actio
System.pr
};
}
Anonymous new ActionListener() can be replaced with lambda
Replace with lambda Alt+Shift+Enter
More actions... Alt+Enter
java.awt.event
public interface ActionListener
extends java.util.EventListener
The listener interface for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked.
Since: 1.1

Figure 9-3. IntelliJ offering lambda refactoring

9.2 Using Predefined Lambda Interfaces or Rolling Your Own

Problem

You want to use existing interfaces, instead of defining your own, for use with lambdas.

Solution

Use the Java 8 lambda functional interfaces from `java.util.function`.

Discussion

In Recipe 9.1, we used the interface method `acceptCamera()` defined in the interface `CameraAcceptor`. Acceptor-type methods are quite common, so the package java.util.function includes the `Predicate<T>` interface, which we can use instead of `CameraAcceptor`. This interface has only one method—`boolean test(T t)`:

```
interface Predicate<T> {
    boolean test(T t);
}
```

This package includes about 50 of the most commonly needed functional interfaces, such as `IntUnaryOperator`, which takes one `int` argument and returns an `int` value; `LongPredicate`, which takes one `long` and returns `boolean`; and so on.

To use the `Predicate` interface, as with any generic type, we provide an actual type for the parameter `Camera`, giving us (in this case) the parameterized type `Predicate<Camera>`, which is the following (although we don't have to write this out):

```
interface Predicate<Camera> {
    boolean test(Camera c);
}
```

So now our search application will be changed to offer us the following search method:

```
public List<Camera> search(Predicate p);
```

Conveniently, this has the same signature as our own `CameraAcceptor` from the point of view of the anonymous methods that lambdas implement, so the rest of our code doesn't have to change! This is still a valid call to the `search()` method:

```
results = searchApp.search(c -> c.isIlc() && c.getPrice() < 500);
```

Here is the implementation of the `search` method:

main/src/main/java/functional/CameraSearchPredicate.java

```
  public List<Camera> search(Predicate<Camera> tester) {
      List<Camera> results = new ArrayList<>();
      privateListOfCameras.forEach(c -> {
          if (tester.test(c))
              results.add(c);
      });
      return results;
  }
```

Suppose we only need the list to do one operation on each element, and then we'll discard it. Upon reflection, we don't actually need to get the list back; we merely need to get our hooks on each element that matches our `Predicate` in turn.

Roll Your Own Functional Interface

While the JDK provides a good set of functional interfaces, there may be cases where you'd want to create your own. This is a simple example of a functional interface:

```
@FunctionalInterface
interface MyFunctionalInterface {
    int compute(int x);
}
```

The `@FunctionalInterface` annotation tells the compiler to ensure that a given interface is and remains functional. Its use is analogous to `@Override` (both annotations are in `java.lang`). It is always optional.

`MyFunctionalInterface` could be used to process an array of integers, like this:

```
  static int[] integers = {1, 2, 3};

  public static void main(String[] args) {
    int total = 0;
    for (int i : integers)
      total += process(i, x ->  x * x + 1);
    System.out.println("The total is " + total);
  }

  private static int process(int i, MyFunctionalInterface o) {
    return o.compute(i);
  }
```

If `compute` were a nonfunctional interface—having multiple abstract methods—you would not be able to use it in this fashion.

Sometimes, of course, you really do need an interface to have more than one method. In that case, the illusion (or the effect) of functionality can sometimes be preserved by denoting all but one of the methods with the `default` keyword—the nondefault method will still be usable in lambdas. A `default` method has a method body:

```
public interface ThisIsStillFunctional {
    default int compute(int ix) { return ix * ix + 1 };
    int anotherMethod(int y);
}
```

Only `default` methods may contain executable statements, and there may only be one nondefault method per functional interface.

By the way, the `MyFunctionalInterface` given earlier can be totally replaced by `java.util.function.IntUnaryOperator`, changing the method name `apply()` to `applyAsInt()`. There is a version of the `ProcessInts` program under the name `ProcessIntsIntUnaryOperator` in the *javasrc* repository.

`Default` methods in interfaces can be used to produce *mixins*, as described in Recipe 9.9.

9.3 Simplifying Processing with Streams

Problem

You want to process some data through a pipeline-like mechanism.

Solution

Use a `Stream` class and its operations.

Discussion

Streams are a new mechanism introduced with Java 8 to allow a collection to send its values out one at a time through a pipeline-like mechanism where they can be processed in various ways, with varying degrees of parallelism. There are three types of methods involved with Streams:

- Stream-producing methods (see Recipe 7.4).
- Stream-passing methods, which operate on a Stream and return a reference to it, in order to allow for *fluent programming* (chained method calls); examples include `distinct()`, `filter()`, `limit()`, `map()`, `peek()`, `sorted()`, and `unsorted()`.
- Stream-terminating methods, which conclude a streaming operation; examples include `collect()`, `count()`, `findFirst()`, `max()`, `min()`, `reduce()`, and `sum()`.

In Example 9-2, we have a list of `Hero` objects representing (mostly) heroic individuals through the ages. We use the `Stream` mechanism to filter just the adult heroes and then sum their ages. We use it again to sort the heroes' names alphabetically.

In both operations we start with a stream generator (`Arrays.stream()`); we run it through several steps, one of which involves a mapping operation. Mapping in this context, not to be confused with `java.util.Map`, transforms the stream data from one form to another as it is sent along the pipeline. An example of this is mapping a `Person` object to the person's age (see `mapToInt` in Example 9-2). The stream is wrapped up by a terminal operation. The map and filter operations almost invariably are controlled by a lambda expression (inner classes would be too tedious to use in this style of programming!).

Example 9-2. main/src/main/java/functional/SimpleStreamDemo.java

```
static Hero[] heroes = {
  new Hero("Grelber", 21),
  new Hero("Roderick", 12),
  new Hero("Thor", 35),
```

```
    new Hero("Superman", 65),
    new Hero("Batman", 37),
    new Hero("Palladin", 50),
    new Hero("Iron Man", 42),
    new Hero("Athena", 50)
  };

  public static void main(String[] args) {

    long adultYearsExperience = Arrays.stream(heroes)
        .filter(b -> b.age >= 18)
        .mapToInt(b -> b.age)
        .sum();
    System.out.println("We're in good hands! The adult superheros have " +
        adultYearsExperience + " years of experience");

    List<Object> sorted = Arrays.stream(heroes)
        .sorted((h1, h2) -> h1.name.compareTo(h2.name))
        .map(h -> h.name)
        .collect(Collectors.toList());
    System.out.println("Heroes by name: " + sorted);
  }
```

And let's run it to be sure it works:

```
We're in good hands! The adult superheros have 300 years of experience
Heroes by name: [Athena, Batman, Grelber, Iron Man, Palladin,
        Roderick, Superman, Thor]
```

See the Javadoc for the `java.util.stream.Stream` interface for a complete list of the operations.

9.4 Simplifying Streams with Collectors

Problem

You construct Streams, but it's hard to get the results into a form you want.

Solution

Consider using a `Collector` to terminate your stream.

Discussion

Example 9-2 ended the second half with a call to `collect()`. The argument to `collect()` is of type `Collector`, which this recipe considers in more detail. `Collectors` are a form of what classical FP languages call *folds*. Folds are also called reduce, accumulate, aggregate, compress, or inject operations. A fold in functional programming is a terminal operation, analogous to collapsing a whole string of tickets into a flat

pile (see Figure 9-4). The string of tickets represents the `Stream`, the folding operation is represented by a function, and the final result is, well, all folded up. It will often include a combining operation, analogous to counting the tickets as they are folded.

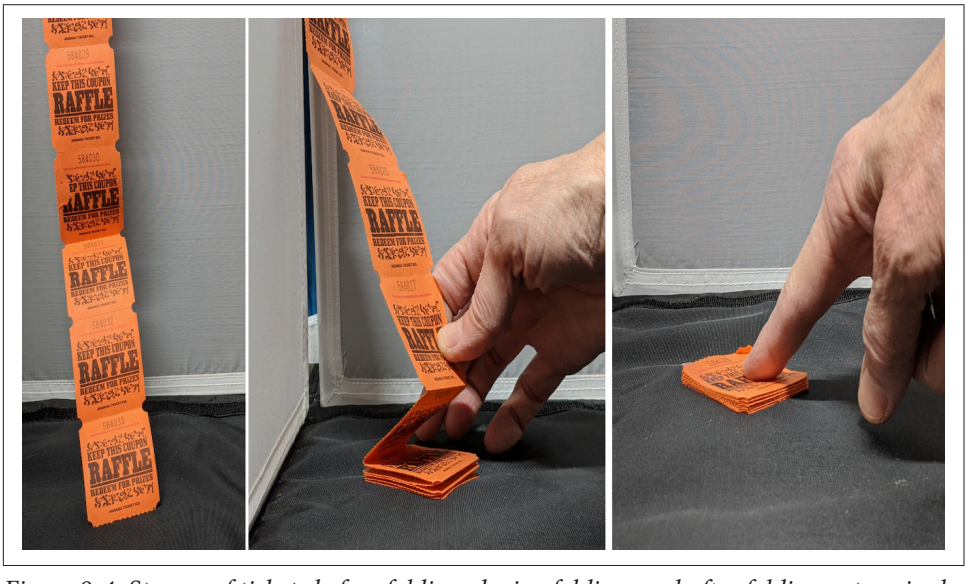


Figure 9-4. Stream of tickets before folding, during folding, and after folding: a terminal operation

Note that in the first panel of Figure 9-4 we don't know how long the `Stream` is, but we expect that it will terminate eventually.

Collector as used in Java refers to a terminal function that analyzes/summarizes the content of a `Stream`. Technically, `Collector` is an interface whose implementation is specified by three (or four) functions that work together to accumulate entries into a `Collection` or `Map` or other mutable result container, and optionally a final transform on the result. The functions are as follows:

`supplier()`
: Creates a new result container

`accumulator()`
: Adds a new data element into the result container

`combiner()`
: Combines two result containers into one

`finisher()` *(optional)*
: Performs a final transform on the container

While you can easily compose your own `Collector` implementation, it is often expedient to use one of the many useful implementations predefined in the `Collectors` class. Here are a couple of simple examples:

```
    int howMany = cameraList.stream().collect(Collectors.counting());
    double howMuch = cameraList.filter(desiredFilter).
            collect(Collectors.summingDouble(Camera::getPrice);
```

The `::` syntax is explained in Recipe 9.8; the effect here is to replace each `Camera` object in the Stream with the result of calling its `getPrice()` method.

In Example 9-3 I implement the classic *word frequency count* algorithm: take a text file, break it into individual words, count the occurrence of each word, and list the *n* most used words, sorted by frequency in descending order.

In Unix terms this could be implemented (assuming $n = 20$) as:

```
    prep $file | sort | uniq -c | sort -nr | head -20
```

where `prep` is a script that uses the Unix tool `tr` to break lines into words and turn the words into lowercase.

Example 9-3. main/src/main/java/functional/WordFreq.java

```
/**
 * Implement word frequency count; shown in two main steps
 * (WordFreq2 shows the steps combined into one statement).
 */
public class WordFreq {
  public static void main(String[] args) throws IOException {

    String fileName = args.length == 0 ?  "README.adoc" : args[0];

    // Collect words with a mutable reduction into
    // a Map<String,Long>
    Map<String,Long> map = Files.lines(Path.of(fileName))
      .filter(line->!line.isEmpty())
      .flatMap(s -> Stream.of(s.split(" ")))
      .collect(Collectors.groupingBy(
        String::toLowerCase, Collectors.counting()));

    // Get map's entrySet as a stream,
    // sort -r, count, print highest 20.
    map.entrySet().stream()
      .sorted(Map.Entry.<String,Long>comparingByValue()
      .reversed())
      .limit(20)
      .map(entry -> "%4d %s".formatted(entry.getValue(), entry.getKey()))
      .forEach(System.out::println);
  }
}
```

There are two steps. First, create a map of the words and their frequencies. Second, sort these in reverse order, stop at number 20, format them neatly, and print.

The first part uses `Files.lines()` from Chapter 10 to get a `Stream` of `Strings`, which is broken into individual words using the `Stream` method `flatMap()` combined with the `String` method `split()` to break on one or more spaces. We use `flatmap()` here because `map()` would produce a stream of `<List<String>>`, whereas `flatmap()` produces the desired `Stream` of `Strings`. The result is collected into a map using a `Collector`. I had initially used a homemade collector:

```
.collect(HashMap::new, (m,s)->m.put(s, m.getOrDefault(s,0)+1), HashMap::putAll);
```

This form of `collect()` takes three arguments:

- A `Supplier<R>` or factory method to create an empty container; here I'm just using the `HashMap` constructor.
- An `Accumulator` of type `BiConsumer<R,? super T>` to add each element into the map, adding one each time the same word is found.
- A `Combiner` of type `BiConsumer<R,R> combiner)` to combine all the collections used.

In the case of parallel streams (see Recipe 9.7), the `Supplier` may be called multiple times to create multiple containers, and each part of the Stream's content will be handled by one `Accumulator` into one of the containers. The `Combiner` will merge all the containers into one at the end of processing.

However, Sander Mak pointed out that it's easier to use the existing `Collectors` class's predefined collector `groupingBy`, combining the `toLowerCase()` call and the `collect()` call with this:

```
.collect(Collectors.groupingBy(String::toLowerCase, Collectors.counting()));
```

To further simplify the code, you could combine the two statements into one by doing the following:

- Removing the return value and assignment `Map<String,Long> map =`
- Removing the semicolon from the end of the `collect` call
- Removing the `.map()` from the `entrySet()` call

Then you can say you've implemented something useful in a single Java statement!

9.5 Simplifying Streams with Stream Gatherers 22P

Problem

You are composing a `Stream` but find that the intermediate operations cannot be expressed succinctly. You realize that nothing in the material covered so far will help simplify them.

Solution

Use a `Stream Gatherer`, either one that is built-in or one of your own construction.

Discussion

The Streams API has been quite successful in allowing the creative composition of applications. As pointed out in Recipe 9.3, a stream consists of one generational step, a number of intermediate steps, and one terminal step. However, the set of medial operations is finite. Suppose we have a class full of graduate students whom we want to set to work researching some aspect of life in countries around the world. Assume that each student can handle, say, 10 countries in one term. We have the ISO 3166 two-letter country codes saved in a text file. Our goal is to print the list of 10 countries to hand out to each student.

Up to and including Java 21, there was no way to implement this in Streams without writing some extra code. However, in Java 22 and later, we can use a `Stream` with the new `Stream Gatherer` mechanism to do this easily, as shown in Example 9-4.

Example 9-4. main/src/main/java/functional/StreamGathererDemo.java

```
Files.lines(Path.of("country_codes.txt"))
  .gather(Gatherers.windowFixed(NUM_COUNTRIES))
  .limit(5)  // Just sample
  .forEach(System.out::println);
```

When we run this, it "just works." We stopped it at five rows just to prove that it works:

```
$ java StreamGathererDemo.java
["AA", "AB", "AC", "AD", "AE", "AF", "AG", "AH", "AI", "AJ"]
["AK", "AL", "AM", "AN", "AO", "AP", "AQ", "AR", "AS", "AT"]
["AU", "AV", "AW", "AX", "AY", "AZ", "BA", "BB", "BC", "BD"]
["BE", "BF", "BG", "BH", "BI", "BJ", "BK", "BL", "BM", "BN"]
["BO", "BP", "BQ", "BR", "BS", "BT", "BU", "BV", "BW", "BX"]
$
```

As is common, a utility class is named in the plural, e.g., `Gatherers`. There are five predefined gatherers, obtainable from these static methods in the `Gatherers` class:

`fold`
: A many-to-one gatherer; see the fold operation in Recipe 9.4

`mapConcurrent`
: Like regular `Stream.map()` but lets you specify a maximum level of concurrency, which is implemented using virtual threads (Recipe 11.2)

`scan`
: Outputs the result of a given operation on each item in the input stream (see `StreamGathererScanDemo` in *javasrc*)

`windowFixed`
: Groups into fixed-size buckets, as shown in Example 9-4

`windowSliding`
: Groups into sequential permutations, e.g, given the integers from 1 up as input, `Gatherer.windowSliding(3)` would output [1,2,3], [2,3,4], [3,4,5], [4,5,6], etc.

9.6 Simplifying Streams with Your Own Stream Gatherer 22P

Problem

There doesn't seem to be a stream gatherer that does quite what you want.

Solution

Write your own! `Gatherer` is just an interface.

Discussion

You can implement your own `Stream Gatherer`. A `Gatherer` consists of up to four functions:

- The initializer function (optional) creates an object to maintain any private state needed during processing.
- The integrator function (required) processes or "integrates" each element from the input stream. It can use the private state object from the initializer function to determine whether or not to output the current input item to the output stream. It can even stop the processing before reaching the end of the input stream, if the logic so dictates.
- The combiner function (optional) can evaluate the gatherer in parallel if the input stream is parallel. This can be used in cases where an operation is ordered in nature and can't be parallelized.

- The finisher function (optional) is invoked when there are no more input elements to consume. This function can inspect the private state object and choose to emit additional (leftover?) output elements, throw an exception if requirements were not met by any or all of the input elements, or simply do nothing.

Suppose we have a list of sale amounts and want to use streams to get a list that is distinct, but only for sales that differ by a rounding error amount (less than one cent). For this we could use various stream operators, including `filter()`, but this makes a good demo of how simple a custom `Gatherer` implementation can be. Example 9-5 shows an implementation of this requirement.

Example 9-5. main/src/main/java/functional/StreamGathererCustom.java

```
// Mutable State
class DoubleHolder {
  Double value = 0.0D;
}

// Threshold for distinctness - anything over a penny is distinct
double THRESHOLD = 0.01D;

// The star of the show: a simple custom Gatherer
Gatherer<Double, DoubleHolder, Double> distinctByAmount =
  Gatherer.ofSequential(

    // Initializer
    DoubleHolder::new,

    // Integrator
    (state, element, downstream) -> {
      var distinct = Math.abs(element - state.value) > THRESHOLD;
      state.value = element;
      return distinct ? downstream.push(element) : !downstream.isRejecting();
    }
);

// Sample data
List<Double> sales = List.of(
  125.00,
  125.01,
  125.015,
  250.00,
  75.00
);

void main() {
  System.out.println("Raw");
  sales.stream().sorted().forEach(System.out::println);

  System.out.println("Distinct");
```

```
  sales.stream()
    .sorted()
    .gather(distinctByAmount)
    .forEach(System.out::println);
}
```

The key component is the `Gatherer`, defined using the factory method `Gatherer.of()`. There are many overloads of this, but since we're only going to run a small number of transactions each time, we don't need parallelism, so we don't need the optional combiner function. And, each distinct value is pushed downstream as we process it, so we don't even need the optional finisher function. The initializer just constructs an instance of our state class (`DoubleHolder`). The integrator takes each item and compares it to the previous one; if they differ enough, the item is pushed downstream. The integrator method returns `boolean` to indicate whether more elements are to be consumed or not; if the downstream is already in rejecting mode, there is no point in pushing any more elements, so the `Gatherer` will shut down.

Note that in the definitions of gatherers, the type paramater `TR` appears; this indicates that the same type applies both to values consumed (`T` for type of input) and sent downstream (`R` for returned) by this gatherer.

See Also

There is more to the subject of `Gatherers` than we can cover in these two recipes. See the `Gatherer` documentation. Another tutorial is on Ben Weidig's blog.

9.7 Improving Throughput with Parallel Streams and Collections

Problem

You want to combine `Streams` with parallelism and still be able to use the non-thread-safe Collections API.

Solution

Use a parallel stream.

Discussion

To enable efficient operations, *parallel streams* let you use the non-thread-safe collections safely, as long as you do not modify the collection while you are operating on it.

To use a parallel stream, you just ask the collection for it, using `parallelStream()` instead of the `stream()` method we used in Recipe 9.3. This does not guarantee a

speed-up, though it may be faster: to be sure, you need to profile the code with a reasonable-sized volume of data; see Recipe 2.16.

For example, suppose that our camera business takes off, and we need to find cameras by type and price range *quickly*, and with less code than we used before (see Example 9-6).

Example 9-6. src/main/java/functional/CameraSearchParallelStream.java

```
public static void main(String[] args) {
  System.out.println("Search Results using For Loop");
  for (Object camera : listOfCameras.parallelStream(). ❶
      filter(c -> c.isIlc() && c.price() < 500).       ❷
      toArray()) {                                     ❸
    System.out.println(camera);                        ❹
  }

  System.out.println(
    "Search Results from shorter, more functional approach");
  listOfCameras.parallelStream().                      ❺
              filter(c -> c.isIlc() && c.price() < 500).
      forEach(System.out::println);
}
```

❶ Create a parallel stream from the `List` of `Camera` objects. The end result of the stream will be iterated over by the for-each loop.

❷ Filter the cameras on price, using the same `Predicate` lambda that we used in Recipe 9.1.

❸ Terminate the `Stream` by converting it to an array.

❹ The body of the for-each loop: print one `Camera` from the `Stream`.

❺ A more concise way of writing the search.

This is only reliable as long as no thread is modifying the data at the same time as the searching is going on. See the thread interlocking mechanisms in Chapter 11 to see how to ensure this.

9.8 Using Existing Code as Functional with Method References

Problem

You have existing code that matches a functional interface and want to use it without renaming methods to match the interface name.

Solution

Use function references such as `MyClass::myFunc` or `someObj::someFunc`.

Discussion

The word *reference* is almost as overloaded in Java as the word *session*. Consider the following:

- Ordinary objects are usually accessed with references.
- Reference types such as `WeakReference` have defined semantics for garbage collection.
- And now, for something completely different, Java 8+ lets you reference an individual method.

The *method reference* syntax consists of an object reference or class name, two colons, and the name of a method that can be invoked in the context of the object or class name (as per the usual rules of Java, a class name can refer to static methods and an instance can refer to an instance method). To refer to a constructor as the method, you can use `new`—for example, `MyClass::new`. The reference creates a lambda that can be invoked, stored in a variable of a functional interface type, and so on.

In Example 9-7, we create a `Runnable` reference that holds, not the usual `run` method, but rather a method with the same type and arguments but with the name `walk`. Note the use of `this` as the object part of the method reference. We then pass this `Runnable` into a `Thread` constructor and start the thread, with the result that `walk` is invoked where `run` would normally be.

Example 9-7. main/src/main/java/functional/ReferencesDemo.java

```
/** "Walk, don't run" */
public class ReferencesDemo {

  // Assume this is an existing method we don't want to rename
  public void walk() {
    System.out.println("ReferencesDemo.walk(): Stand-in run method called");
```

```
  }

  // This is our main processing method; it runs "walk" in a Thread
  public void doIt() {
    Runnable r = this::walk;
    new Thread(r).start();
  }

  // The usual simple main method to start things off
  public static void main(String[] args) {
    new ReferencesDemo().doIt();
  }
}
```

The output is as follows:

```
    ReferencesDemo.walk(): Stand-in run method called
```

The code could be simplified by moving the body of `doit()` into `main()`, but then the `walk()` method would need to be static, and the reference would be `ReferencesDemo::walk`.

Example 9-8 creates an `AutoCloseable` for use in a `try-with-resources`. The normal `AutoCloseable` method is `close()`, but ours is named `cloz()`. The `AutoCloseable` reference variable `autoCloseable` is created inside the `try` statement, so its `close`-like method will be called when the body completes. In this example, we are in a static `main` method wherein we have a reference `rnd2` to an instance of the class, so we use this in referring to the `AutoCloseable`-compatible method.

Example 9-8. main/src/main/java/functional/ReferencesDemo2.java

```
public class ReferencesDemo2 {
  void cloz() {
    System.out.println("Stand-in close() method called");
  }

  public static void main(String[] args) throws Exception {
    ReferencesDemo2 rd2 = new ReferencesDemo2();

    // Use a method reference to assign the AutoCloseable interface
    // variable "ac" to the matching method signature "c" (obviously
    // short for close, but just to show the method name isn't what matters).
    try (AutoCloseable autoCloseable = rd2::cloz) {
      System.out.println("Some action happening here.");
    }
  }
}
```

The output is as follows:

```
Some action happening here.
Stand-in close() method called
```

It is, of course, possible to use this with your own functional interfaces, as defined in Recipe 9.2. You're also probably at least vaguely aware that any normal Java object reference can be passed to `System.out.println()` and you'll get some description of the referenced object. Example 9-9 explores these two themes. We define a functional interface imaginatively known as `FunInterface` with a method with a bunch of arguments (merely to avoid it being mistaken for any existing functional interface). The method name is `process`, but as you now know the name is not important; our implementation method goes by the name `work`. The `work` method is static, so we could not state that the class implements `FunInterface` (even if the method names were the same, a static method may not hide an inherited instance method), but we can nonetheless create a lambda reference to the `work` method. We then print this out to show that it has a valid structure as a Java object.

Example 9-9. main/src/main/java/functional/ReferencesDemo3.java

```
public class ReferencesDemo3 {

  interface FunInterface {
    void process(int i, String j, char c, double d);
  }

  public static void work(int i, String j, char c, double d){
    System.out.println("In work " + c);
  }

  public static void main(String[] args) {
    FunInterface sample = ReferencesDemo3::work;
    sample.process(42, "Hello", '\u263A', Math.PI);
    System.out.println("My process method is " + sample);
  }
}
```

This generates the following output:

```
In work ☺
My process method is functional.ReferencesDemo3$$Lambda/
0x00000cde2414c3f8@6591f517
```

The `$Lambda` in the name is structurally similar to the `$1` used for anonymous inner classes.

The fourth way, to use a function reference, an instance method of an arbitrary object of a particular type, may be the most esoteric. It allows you to declare a reference to an instance method but without specifying which instance. Because there is no particular instance in mind, you again use the class name. This means you can use it with

any instance of the given class! In Example 9-10, we have an array of `Strings` to sort. Because the names in this array can begin with a lowercase letter, we want to sort them using the `String` method `compareToIgnoreCase()`, which nicely ignores case differences for us.

Because I want to show the sorting several different ways, I set up two array references: the original, unsorted reference, and a working reference that is re-created, sorted, and printed using a simple dump routine, which isn't shown (it's just a `for` loop printing the strings from the passed array).

Example 9-10. main/src/main/java/functional/ReferencesDemo4.java

```
static final List<String> unsortedNames = List.of(
  "Gosling", "de Raadt", "Amdahl", "Turing", "Ritchie", "Hopper"
);

public static void main(String[] args) {
  List<String> names;

  // Sort using
  // "an Instance Method of an Arbitrary Object of a Particular Type"
  names = new ArrayList<>(unsortedNames);
  Collections.sort(names, String::compareToIgnoreCase);    ❶
  dump(names);

  // Equivalent Lambda:
  names = new ArrayList<>(unsortedNames);
  Collections.sort(names,
    (str1, str2) -> str1.compareToIgnoreCase(str2));       ❷
  dump(names);

  // Equivalent old way:
  names = new ArrayList<>(unsortedNames);
  Collections.sort(names, new Comparator<String>() {       ❸
    @Override
    public int compare(String str1, String str2) {
      return str1.compareToIgnoreCase(str2);
    }
  });
  dump(names);

  // Simpest way, using existing comparator
  names = new ArrayList<>(unsortedNames);
  Collections.sort(names, String.CASE_INSENSITIVE_ORDER); ❹
  dump(names);
}
```

❶ Using an instance method of an arbitrary object of a particular type declares a reference to the `compareToIgnoreCase` method of any `String` used in the invocation.

❷ Shows the equivalent lambda expression.

❸ Shows "Your grandparents' Java" way of doing things.

❹ Using the exported `Comparator` directly, just to show that there is always more than one way to do things.

Just to be safe, I ran the demo and got the expected output:

```
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
Amdahl, de Raadt, Gosling, Hopper, Ritchie, Turing
```

9.9 Java Mixins: Mixing in Methods

Problem

You've heard about mixins and want to apply them in Java.

Solution

Use static imports. Or, declare one or more functional interfaces with a `default` method containing the code to execute, and simply implement it.

Discussion

Developers from other languages sometimes deride Java for its inability to handle mixins, the ability to mix in bits of code from other classes.

One way to implement mixins is with the *static import* feature, which has been in the language for a decade. This is often done in unit testing (see Recipe 2.9). A limitation of this approach is that, as the name implies, the methods must be static methods, not instance methods.

A newer mechanism depends on an interesting bit of fallout from the Java 8 language changes in support of lambdas: you can now mix in code from unrelated places into one class. Has Java finally abandoned its staunch opposition to multiple inheritance? It may seem that way when you first hear it, but relax: you can only pull methods from multiple interfaces, not from multiple classes. If you didn't know that you could have methods defined (rather than merely declared) in interfaces, see "Subclass, Abstract Class, or Interface?" on page 300. Consider the following example:

main/src/main/java/lang/MixinsDemo.java

```
interface Bar {
  default String filter(String s) {
    return "Filtered " + s;
  }
}

interface Foo {
  default String convolve(String s) {
    return "Convolved " + s;
  }
}

public class MixinsDemo implements Foo, Bar{

  public static void main(String[] args) {
    String input = args.length > 0 ? args[0] : "Hello";
    String output = new MixinsDemo().process(input);
    System.out.println(output);
  }

  private String process(String s) {
    return filter(convolve(s)); // methods mixed in!
  }
}
```

If we run this, we see the expected results:

```
C:\javasrc>javac -d build lang/MixinsDemo.java
C:\javasrc>java -cp build lang.MixinsDemo
Filtered Convolved Hello

C:\javasrc>
```

Presto—Java now supports mixins! And if there are conflicting methods, please refer back to "Default methods" on page 305.

Does this mean you should go crazy trying to build interfaces with code in them? No. Remember this mechanism was designed to do the following:

- Provide the notion of functional interfaces for use in lambda calculations.
- Give the ability to retrofit interfaces with new methods, without having to change *old* implementations. As with many changes made in Java over the years, backward compatibility was a huge driver.

Used sparingly, functional interfaces can provide the ability to mix in code to build up applications in another way than direct inheritance, aggregation, or AOP. Overused, it can make your code heavy, drive pre–Java 8 developers crazy, and lead to chaos.

9.10 Functional Programming with Flow and Reactive Streams

Problem

You have a lot of different I/O that could be handled asynchronously, and you want to try working with the *reactive* style of programming.

Solution

Use the Reactive API in `java.util.concurrent.Flow`, or use a third-party implementation.

Discussion

The original Reactive Manifesto states that reactive systems "are software architectures that are responsive, resilient, elastic and message driven." The last attribute implies a publish-subscribe model; the former three imply a form in which one (or more) publishers push data to one or more subscribers, each of whom can control how rapidly data gets sent to them. This notion of controlling the data rate is often called *back pressure*, in the sense that it exerts pressure on the sender to reduce the flow. Back pressure is new and significant, as it can allow the receiver to inform the sender when the latter is sending too quickly, for example. In this API, back pressure is controlled very simply, by the subscriber client code calling a `request()` method on the `Subscriber`, both on connection and after every receipt of data.

The class `java.util.concurrent.Flow` exports two interfaces, called `Publisher` and `Subscriber`. `Subscriber` can be implemented directly. The `Publisher` has more to do, and users of this API will often use the `SubmissionPublisher` implementation class directly. Example 9-11 shows this, with one `Publisher` and two `Subscribers`.

Example 9-11. main/src/main/java/functional/FlowDemo.java

```
public class FlowDemo {

  public static void main(String[] args) throws InterruptedException {

    // This is a Flow.Publisher implementation that handles submission/
    // forwarding, and has a close() method.
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

    // Our implementation of the Subscriber interface
    class MySubscriber implements Flow.Subscriber<String> {
      private Flow.Subscription subscription;
      private String id;
```

```
    MySubscriber(String id) {
      this.id = id;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
      this.subscription = subscription;
      subscription.request(getNumBuffers());
    }

    @Override
    public void onNext(String item) {
      System.out.println(id + ": Received: " + item);
      subscription.request(getNumBuffers());
    }

    @Override
    public void onError(Throwable throwable) {
      throwable.printStackTrace();
    }

    @Override
    public void onComplete() {
      System.out.println(id + ": Done");
    }
  };
  Flow.Subscriber<String> subscriber = new MySubscriber("1");
  publisher.subscribe(subscriber);
  Flow.Subscriber<String> subscriber2= new MySubscriber("2");
  publisher.subscribe(subscriber2);
  for (String mesg : messages) {
    System.out.println("Sending: " + mesg);
    publisher.submit(mesg);
  }
  publisher.close();
  Thread.sleep(1000);
}

/**
 * Compute the number of buffers we want to receive, to
 * manage "back pressure" - so we don't get sent more than
 * we can handle.
 * A placeholder for now - "1" will always work.
 */
static int getNumBuffers() {
  return 1;
}

static String[] messages = {
  "Hello, world of Java!",
  "Hope you're having a good day!",
```

```
    "Things are really hopping now",
    "The clock is ticking.",
    "Day is done. Goodbye!",
  };
}
```

Here is the output of running Example 9-11:

```
$ java FlowDemo.java
Sending: Hello, world of Java!
Sending: Hope you're having a good day!
Sending: Things are really hopping now
Sending: The clock is ticking.
Sending: Day is done. Goodbye!
2: Received: Hello, world of Java!
1: Received: Hello, world of Java!
2: Received: Hope you're having a good day!
1: Received: Hope you're having a good day!
2: Received: Things are really hopping now
1: Received: Things are really hopping now
2: Received: The clock is ticking.
1: Received: The clock is ticking.
2: Received: Day is done. Goodbye!
1: Received: Day is done. Goodbye!
1: Done
2: Done
$
```

See Also

There is much more to the world of reactive programming. Besides the Java 9 `Flow` described here, RxJava and Akka-Streams are two more comprehensive Java application frameworks. Android users can combine RxJava and RxAndroid. Flutter developers can use RxDart. React Native is a complete JavaScript-based toolkit for building reactive apps for desktop and mobile. Baeldung's website and the article "Reactive Programming with Java 9's Flow" by Moisés Macero both contain extended Java examples.

CHAPTER 10

Input and Output: Reading, Writing, and Directory Tricks

10.0 Introduction

Most programs need to interact with the outside world, and one common way of doing so is by reading and writing files. Files are usually stored on some persistent medium such as a disk drive; and, for the most part, we shall happily ignore the differences between files stored on a hard disk (and all the operating system–dependent filesystem types), a USB drive or SD card, a DVD-ROM, and other storage devices. For us, they're just files. And, like most other languages and OSes, Java extends the reading-and-writing model to network (socket) communications, which we'll touch on in Chapters 14 and 15.

Java provides many classes for input and output, as summarized later in Figure 10-1. This chapter covers all the normal input/output operations such as opening/closing and reading/writing files. Files are assumed to reside on some kind of file store or permanent storage. Distributed filesystems such as Apache Hadoop Distributed File System (HDFS), Sun's Network File System (NFS, common on Unix and available for Windows), SMB (the Windows network filesystem, available for Unix via the open source Samba program), and Filesystem in User SpacE (FUSE; implementations available for most Unix/Linux/macOS systems) are assumed to work just like disk filesystems, except where noted.

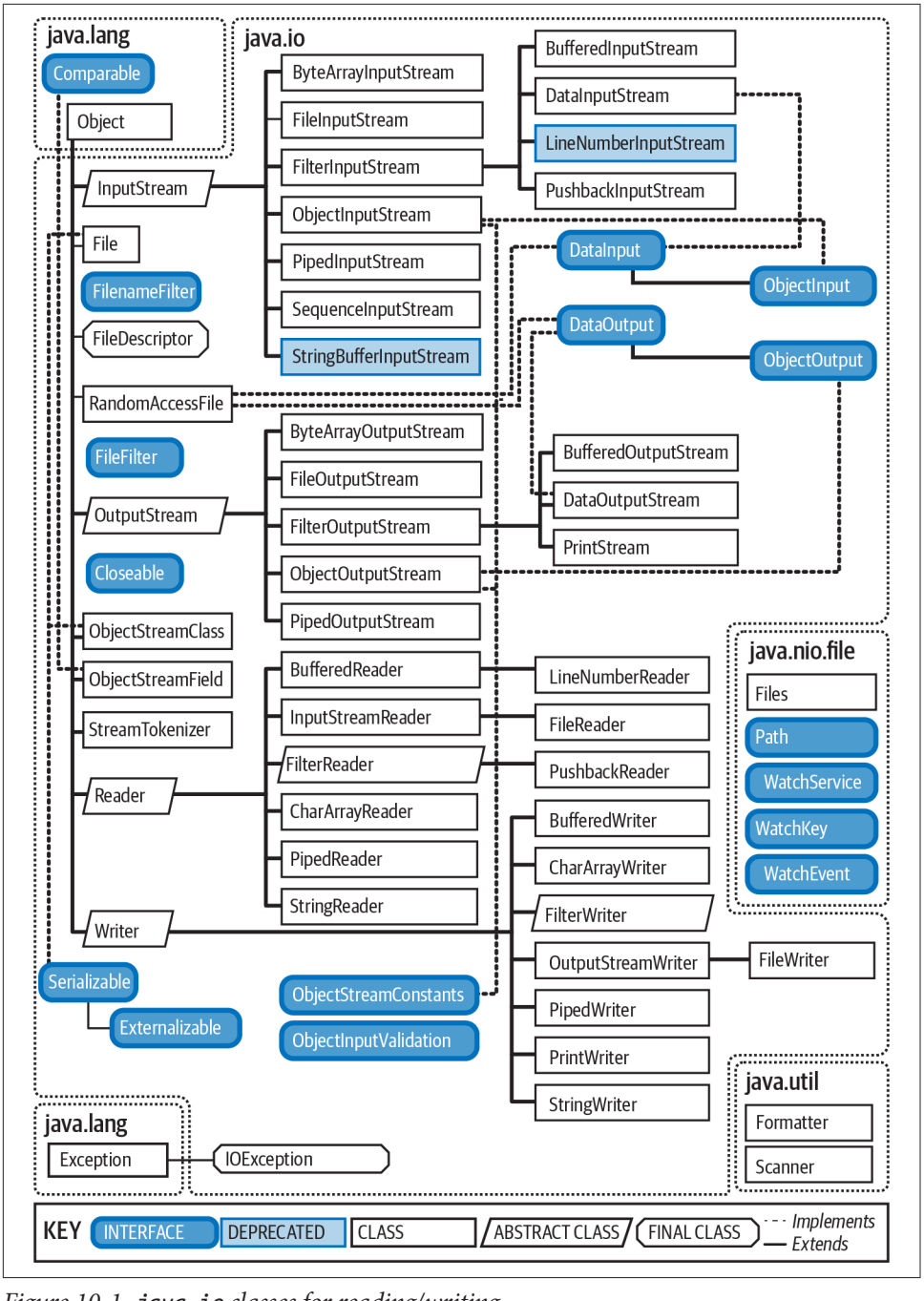


Figure 10-1. `java.io` classes for reading/writing

The support for reading and writing is in two major parts:

- The `InputStream/OutputStream/Reader/Writer` classes—the traditional methods of reading/writing files—have been largely unchanged since Java 1.0 and 1.1. Modern Java provides a new class, `java.nio.file.Files`.
- All modern operating systems provide the means to organize groups of files into directories, or folders. This chapter covers directories: how to create them and how to navigate them. The `Files` class provides most of the support for processing directories, but it also introduces a number of convenience routines for easily reading, writing, and copying files, most of which are covered in this chapter. These are generally more convenient than using the traditional I/O classes. We cover both in this chapter.

The older `java.io.File` command is out of favor; best practice recommends using `Files` in its stead. Examples of the older `File` are left in the *javasrc* source repository for those who need them.

The first part of the chapter is largely devoted to the `Files` and `Path` classes in `java.nio.file`. These two classes provide the ability to list directories, obtain file status, rename and delete files on disk, create directories, and perform other filesystem operations. They also provide the ability to read a file line by line into a `Stream<String>`. These two classes together largely supplant the older `java.io.File` class. They were introduced in Java 7, so very little new code should use the older `File` class.

Note that many of the methods of this class attempt to modify the permanent file store, or disk filesystem, of the computer on which you run them. Naturally, you might not have permission to change certain files in certain ways. If you don't have permission to perform the attempted operation, as detected by the underlying operating system, Java will throw an instance of the checked exception `IOException`. This must be caught (or declared in the `throws` clause) in any code that calls any method that tries to change the filesystem, and some methods that try to access the filesystem's contents.

There are (at least!) two different uses of the term *stream*. The first is for a stream of bytes to be read or written, and it is covered in this chapter. The second meaning refers to the streams mechanism (`Stream` interface), which is used in modern Java to refer to a connection among cooperating methods, discussed in Recipe 9.3. I'll try to keep these meanings straight by only using `InputStream` and/or `OutputStream` for the former, and `Stream` for the latter.

To give you control over the format of data that you read and write, the `Formatter` and `Scanner` classes provide formatting and scanning operations. `Formatter` allows many formatting tasks to be performed either into a `String` or to almost any output

destination. `Scanner` parses many kinds of objects, again either from a `String` or from almost any input source. These are fairly powerful; each is given its own recipe in this chapter.

10.1 Discovering Filesystem Paths

Problem

You need a `Path` object to work with the `Files` class.

Solution

Use the methods of `Path` to obtain an instance.

Discussion

A `Path` object represents a path into the filesystem, that is, a set of directories starting at the root of the filesystem, and possibly a file, like *C:\Users\user\Downloads* or */home/ian/Downloads*. The filesystem object does not need to exist on disk at the time you create a `Path` representing it. The `Files` class can tell you whether the filesystem object represented by a given `Path` exists, and:

- If not, can bring that `Path` into being as a file or as a directory, open it for writing, etc.
- If so, can read the file, change the file's attributes or contents, or even destroy the file.

`Path` objects are easily created with `Path.of(String name)`, which has several overloads.[1] `Path` is an interface whose implementation is provided by a provider class called `Filesystem`. `Path` has many methods, listed in Table 10-1.

Table 10-1. Public static methods in `java.nio.file.Path`

Access	Return type	Method
static	`Path`	`of(String, String…);`
static	`Path`	`of(URI);`
abstract	`FileSystem`	`getFileSystem();`
abstract	`boolean`	`isAbsolute();`
abstract	`Path`	`getRoot();`

1 A `Paths` object exists and has two methods that return `Path` objects, but this class is not recommended and is probably going to be deprecated soon.

Access	Return type	Method
abstract	Path	getFileName();
abstract	Path	getParent();
abstract	int	getNameCount();
abstract	Path	getName(int);
abstract	Path	subpath(int, int);
abstract	boolean	startsWith(Path);
default	boolean	startsWith(String);
abstract	boolean	endsWith(Path);
default	boolean	endsWith(String);
abstract	Path	normalize();
abstract	Path	resolve(Path);
default	Path	resolve(String);
default	Path	resolveSibling(Path);
default	Path	resolveSibling(String);
abstract	Path	relativize(Path);
abstract	URI	toUri();
abstract	Path	toAbsolutePath();
abstract	Path	toRealPath(LinkOption…) throws IOException;
default	File	toFile();
abstract	WatchKey	register(WatchService, WatchEvent$Kind<?>[], WatchEvent$Modifier…) throws IOException;
default	WatchKey	register(WatchService, WatchEvent$Kind<?>…) throws IOException;
default	Iterator<Path>	iterator();
abstract	int	compareTo(Path);
abstract	boolean	equals(Object);
abstract	int	hashCode();
abstract	String	toString();
default	int	compareTo(Object);

Some of these methods are demonstrated in Example 10-1. The `register(WatchService...)` methods are described in Recipe 10.19.

Example 10-1. main/src/main/java/dir_file/PathDemo.java

```
import java.nio.file.Path;

public class PathDemo {
```

```
public static void main(String[] args) {
  var p = Path.of("/etc/hosts");
  display("Path", p);
  display("isAbsolute", p.isAbsolute());
  display("getRoot", p.getRoot());
  display("getParent", p.getParent());
  var q = Path.of("PathDemo.java");
  display("endsWith(.php)", q.endsWith(".php"));
  display("endsWith(.java)", q.endsWith(".java"));
  var s = Path.of("/usr/bin/../../home/ian/../../etc/hosts");
  display("Normalize long path", s = s.normalize());
  display("Equals after normalize", s.equals(p));
  var t = Path.of("Foo.java");
  display("absolutePath", t.toAbsolutePath());
}
```

To work with actual files in a filesystem, `Path` objects are typically used in conjunction with `Files`, as described in Recipe 10.2.

10.2 Getting and Setting File and Directory Information: Files and Path

Problem

You need to know all you can about a given file on disk.

Solution

Use `java.nio.file.Files` methods.

Discussion

The `java.nio.file.Files` class has a plural name both to differentiate it from the legacy `File` class that it replaces and to remind us that it sometimes works on multiple files. There are no public constructors and no instance methods, only static methods in the `Files` class. These methods fall into two categories: informational and operational. The informational methods (see Table 10-2) simply give you information about one file, such as `boolean exists()` or `long size()`. The operational methods (see Table 10-3) either make changes to the filesystem or open a file for reading or writing. Each of the operational methods can throw the checked exception `IOException`; only a few of the informational methods can.

The vast majority of these methods have argument(s) of type `java.nio.file.Path`. A `Path` represents a path into the filesystem, and is described in Recipe 10.1. The `Files` class can tell you whether the file represented by a given `Path` exists, and:

- If not, can bring that `Path` into being as a file or as a directory, open it for writing, etc.
- If so, can read the file, change the file's attributes or contents, or even destroy the file.

`Files` in conjunction with `Path` offers pretty much everything you'd need to write a full-blown file manager application, let alone meet the needs of a more typical application needing file information and/or directory access. The `Files` class has a series of static `boolean` methods that give basic information (see Table 10-2).

Table 10-2. Public static informational methods in `java.nio.file.Files`

Return type	Method	Notes
`boolean`	`exists(Path, LinkOption…);`	
`Object`	`getAttribute(Path, String, LinkOption…);`	
`<V extends FileAttributeView> V`	`getFileAttributeView(Path, Class<V>, LinkOption…);`	
`FileTime`	`getLastModifiedTime(Path, LinkOption…);`	
`UserPrincipal`	`getOwner(Path, LinkOption…);`	
`Set<PosixFile Permission>`	`getPosixFilePermissions(Path, LinkOption…);`	
`boolean`	`isDirectory(Path, LinkOption…);`	
`boolean`	`isExecutable(Path);`	If executable by current user
`boolean`	`isHidden(Path);`	If a "dot file" on Unix, or "hidden" attribute set on some OSes
`boolean`	`isReadable(Path);`	If readable by current user
`boolean`	`isRegularFile(Path, LinkOption… );`	
`boolean`	`isSameFile(Path, Path) throws IOException;`	Has to unwind filesys complexities like `..`, symlinks, …
`boolean`	`isSymbolicLink(Path);`	
`boolean`	`isWritable(Path);`	If writable by current user
`long`	`mismatch(Path, Path);`	
`boolean`	`notExists(Path, LinkOption…);`	

Return type	Method	Notes
`String`	`probeContentType(Path) throws IOException;`	Tries to return MIME type of data
`Path`	`readSymbolicLink(Path) throws IOException;`	
`long`	`size(Path);`	

By "current user" we mean the account under which the current JVM instance is being run.

Most of the methods in Table 10-2 are demonstrated in Example 10-2.

Example 10-2. main/src/main/java/io/FilesInfos.java

```
println("exists", Files.exists(Path.of("/")));
println("isDirectory", Files.isDirectory(Path.of("/")));
println("isExecutable", Files.isExecutable(Path.of("/bin/cat")));
println("isHidden", Files.isHidden(Path.of("~/.profile")));
println("isReadable", Files.isReadable(Path.of("lines.txt")));
println("isRegularFile", Files.isRegularFile(Path.of("lines.txt")));
println("isSameFile", Files.isSameFile(Path.of("lines.txt"),
  Path.of("../main/lines.txt")));
println("isSymbolicLink", Files.isSymbolicLink(Path.of("/var")));
println("isWritable", Files.isWritable(Path.of("/tmp")));
println("isDirectory", Files.isDirectory(Path.of("/")));
println("notexists",
  Files.notExists(Path.of("no_such_file_as_skjfsjljwerjwj")));
println("probeContentType", Files.probeContentType(Path.of("lines.txt")));
println("readSymbolicLink", Files.readSymbolicLink(Path.of("/var")));
println("size", Files.size(Path.of("lines.txt")));
```

Obviously the paths chosen are somewhat system-specific, but when run on my Unix system, the `boolean` methods all returned `true`, and the last three returned this:

```
probeContentType returned text/plain
readSymbolicLink returned private/var
size returned 78
```

Table 10-3 shows the methods that read and/or make changes to filesystem entities.

Table 10-3. Public static operational methods in `java.nio.file.Files`

Return type	Method
`long`	`copy(InputStream, Path, CopyOption…);`
`long`	`copy(Path, OutputStream);`
`Path`	`copy(Path, Path, CopyOption…);`
`Path`	`createDirectories(Path, FileAttribute<?>…);`
`Path`	`createDirectory(Path, FileAttribute<?>…);`

Return type	Method
Path	createFile(Path, FileAttribute<?>…);
Path	createLink(Path, Path);
Path	createSymbolicLink(Path, Path, FileAttribute<?>…);
Path	createTempDirectory(Path, String, FileAttribute<?>…);
Path	createTempDirectory(String, FileAttribute<?>…);
Path	createTempFile(Path, String, String, FileAttribute<?>…);
Path	createTempFile(String, String, FileAttribute<?>…);
void	delete(Path);
boolean	deleteIfExists(Path);
Stream<Path>	find(Path, int, BiPredicate<Path, BasicFileAttributes>, FileVisitOption…);
Stream<String>	lines(Path);
Stream<String>	lines(Path, Charset);
Stream<Path>	list(Path);
Path	move(Path, Path, CopyOption…);
BufferedReader	newBufferedReader(Path);
BufferedReader	newBufferedReader(Path, Charset);
BufferedWriter	newBufferedWriter(Path, Charset, OpenOption…);
BufferedWriter	newBufferedWriter(Path, OpenOption…);
SeekableByteChannel	newByteChannel(Path, OpenOption…);
SeekableByteChannel	newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>…);
DirectoryStream<Path>	newDirectoryStream(Path);
DirectoryStream<Path>	newDirectoryStream(Path, String);
InputStream	newInputStream(Path, OpenOption…);
OutputStream	newOutputStream(Path, OpenOption…);
byte[]	readAllBytes(Path);
List<String>	readAllLines(Path);
List<String>	readAllLines(Path, Charset);
<A extends BasicFileAttributes> A	readAttributes(Path, Class<A>, LinkOption…);
Map<String, Object>	readAttributes(Path, String, LinkOption…);
String	readString(Path);
String	readString(Path, Charset);

Return type	Method
Path	setAttribute(Path, String, Object, LinkOption…);
Path	setLastModifiedTime(Path, FileTime);
Path	setOwner(Path, UserPrincipal);
Path	setPosixFilePermissions(Path, Set<PosixFilePermission>);
Path	write(Path, Iterable<? extends CharSequence>, Charset, OpenOption…);
Path	write(Path, Iterable<? extends CharSequence>, OpenOption…);
Path	write(Path, byte[], OpenOption…);
Path	writeString(Path, CharSequence, Charset, OpenOption…);
Path	writeString(Path, CharSequence, OpenOption…);

To find the information about one file, you can use the informational methods in `Files` and `Path`, as shown in Example 10-3.

Example 10-3. main/src/main/java/dir_file/FileStatus.java (getting file information)

```
public class FileStatus {
  public static void main(String[] argv) throws IOException {

    // Ensure that a filename (or something) was given in argv[0]
    if (argv.length == 0) {
      System.err.println("Usage: FileStatus filename");
      System.exit(1);
    }
    for (String a : argv) {
      status(a);
    }
  }

  public static void status(String fileName) throws IOException {
    System.out.println("---" + fileName + "---");

    // Construct a Path object for the given file.
    Path p = Path.of(fileName);

    // See if it actually exists
    if (!Files.exists(p)) {
      System.out.println("file not found");
      System.out.println();  // Blank line
      return;
    }
    // Print full name
    System.out.println("Canonical name " + p.normalize());
```

```
    // Print parent directory if possible
    Path parent = p.getParent();
    if (parent != null) {
      System.out.println("Parent directory: " + parent);
    }
    // Check if the file is readable
    if (Files.isReadable(p)) {
      System.out.println(fileName + " is readable.");
    }
    // Check if the file is writable
    if (Files.isWritable(p)) {
      System.out.println(fileName + " is writable.");
    }

    // See if file, directory, or other. If file, print size.
    if (Files.isRegularFile(p)) {
      // Report on the file's size and possibly its type
      System.out.printf("File size is %d bytes, content type %s\n",
          Files.size(p),
          Files.probeContentType(p));
    } else if (Files.isDirectory(p)) {
      System.out.println("It's a directory");
    } else {
      System.out.println("I dunno! Neither a file nor a directory!");
    }

    // Report on the modification time.
    final FileTime d = Files.getLastModifiedTime(p);
    System.out.println("Last modified " + d);

    System.out.println();  // blank line between entries
  }
```

When run on Windows with the three arguments shown, it produces this output:

```
C:\javasrc\dir_file>java dir_file.FileStatus   / /tmp/id /autoexec.bat
---/---
Canonical name C:\
File is readable.
File is writable.
Last modified 1970-01-01T00:00:00.00000Z
It's a directory

---/tmp/id---
file not found

---/autoexec.bat---
Canonical name C:\AUTOEXEC.BAT
Parent directory: \
File is readable.
File is writable.
Last modified 2025-10-13T12:43:05.123918Z
File size is 308 bytes.
```

As you can see, the so-called *canonical name* not only includes a leading directory root of *C:*, but also has had the name converted to uppercase. You can tell I ran that on Windows. That version of Windows did not maintain timestamps on directories; the value `0L` gets interpreted as January 1, 1970 (not coincidentally the same time base Unix has used since that time). On Unix, it behaves differently:

```
$ java dir_file.FileStatus / /tmp/id /autoexec.bat
---/---
Canonical name /
File is readable.
It's a directory
Last modified 2025-12-16T01:14:05.226108Z

---/tmp/id---
Canonical name /tmp/id
Parent directory: /tmp
File is readable.
File is writable.
File size is 36768 bytes, content type null
Last modified 2025-12-21T18:46:27.402108Z

---/autoexec.bat---
file not found

$
```

A typical Unix system has no *autoexec.bat* file. And Unix filenames (like those on a Mac) can consist of upper- and lowercase characters: what you type is what you get.

To list the filesystem entries named in a directory, use the `java.nio.file.Files` static method `Stream<Path> list(Path dir)`, passing the `Path` representing the directory. The `java.nio.file.Files` class contains several methods for working with directories. If you just want to list the contents of a directory, use its `list(Path)` method. For example, to list the filesystem entities named in the current directory, just write the following:

```
Files.list(Path.of(".")).forEach(System.out::println);
```

This can become a complete program known as a classless main (see Recipe 1.2), or be used in any release with the following code. Note that on many systems the `Path` objects are returned in the order they occur in the directory, which isn't sorted. In this simple example we use the `Stream.sorted()` method to order the entries alphabetically:

```
public class Ls {
  public static void main(String args[]) throws IOException {
    Files.list(Path.of("."))
      .sorted()
      .forEach(System.out::println);
```

```
    }
  }
```

Of course, there's lots of room for elaboration. You could print the names in multiple columns across the page. Or you could even print down the page because you know the number of items in the list before you print. You could omit filenames with leading periods, as done by the Unix *ls* program. Or you could print the directory names first; I once used a directory lister called *lc* that did this, and I found it quite useful.

If you want to process the directory recursively, you should *not* check each entry to see if it's a file or directory and recurse on directories. Instead, you should use one of the `walk()` or `walkFileTree()` methods discussed in Recipe 10.20; these handle recursion for you. It may be more efficient to use one of the `find()` methods, rather than `walk()` with `filter()`. Example 10-4 shows the `find()` method being used to print all the book source files (names ending in *.asciidoc* or *.adoc*); the callback method is called with the `Path` object and a file attribute descriptor.

Example 10-4. main/src/main/java/dir_file/FindStream.java

```
void main() {
  var dirName = "/home/ian/book";
  try (Stream<Path> paths = Files.find(Path.of(dirName),
    Integer.MAX_VALUE,
    (path, attr) -> {
      return attr.isRegularFile() && (
        path.toString().endsWith(".adoc") ||
        path.toString().endsWith(".asciidoc"));
    })) {
    paths.forEach(System.out::println);
    }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

There is also a set of `Files.newDirectoryStream()` methods, with and without filter callbacks and other arguments, that return a `DirectoryStream<Path>`.

Legacy compatibility File—Files

To use a `Path` with legacy code that needs the older `java.io.File`, simply use `File oldType = Path.toFile()`:

```
jshell> Path p = Path.of("/");
p ==> /

jshell> File f = p.toFile();
f ==> /
```

To go the other way, the `File` class has been retrofitted with a `toPath()` method:

```
jshell> File f = new File("/");
f ==> /

jshell> Path p = f.toPath();
p ==> /
```

Understanding I/O Options: StandardOpenOptions, FileAttribute, PosixFileAttribute, and More

There are several sets of options that can be applied when creating or opening a file, particularly when using the `Files` class. The main option sets that can be applied include those listed in Table 10-4.

Table 10-4. Sets of options

Name	Examples	Usage/notes
`CopyOption`	`StandardCopyOption.REPLACE_EXISTING`	`Files` methods that copy
`LinkOption`	`LinkOption.NOFOLLOW_LINKS`	`Files` methods that write data or read attributes
`FileAttribute`		Name-value pair, used in `Files.create*()` methods
`OpenOption`	`StandardOpenOption.READ,APPEND`	`Files.new{In,Out}putStream()`
`PosixFilePermission`	`OWNER_READ,OTHER_WRITE`	`FileAttribute`
`PosixFilePermissions`	`Set<PosixFilePermission>`	Conversions to/from `rwx` strings

A list of the standard `OpenOption` values is in Table 10-5. These control how a file is to be accessed.

Table 10-5. `OpenOption StandardOpenOption` values

Name	Meaning
`APPEND`	Write at the end of an existing file instead of overwriting it.
`CREATE`	Create the file if it does not exist.
`CREATE_NEW`	Create the file only if it is new; fails with `FileAlreadyExistsException` if file already exists.
`DELETE_ON_CLOSE`	Delete the file when the stream is closed. Useful for temporary files.

Name	Meaning
DSYNC	Write data synchronously, i.e., every write is to be synchronized to disk immediately.
READ	Open the file for reading.
SPARSE	Create as a sparse file, e.g., for random-access writing.
SYNC	Write data and metadata synchronously, i.e., every write or attribute change is to be synchronized to disk immediately.
TRUNCATE_EXISTING	If the file exists, open for writing at the beginning, removing all contents at open time.
WRITE	Open for write access.

POSIX is the IEEE's[2] *P*ortable *O*perating *S*ystem Specification for Un*ix*-like operating systems. Java's `PosixPermission` and its wrapper `PosixPermissions` are used to control who can do what to a file on disk. These are based on the Unix/POSIX permissions laid down in early Unix systems in the early 1970s. There are three actors: owner (which Unix calls user), group, and other (everyone else). Groups is a Unix/POSIX mechanism: a user can be in one or many groups and has permissions based on all the groups they are in; this is an early form of privilege separation.

There are three permissions: read, write, and execute. The latter grants permission to execute a file, but is also used to grant permission to search (list) a directory. For decades these have been expressed as a nine-character permissions string. For example, `rwxr-r--` means the user has read, write, and execute permissions on a given file; other members of the file owner's group have read-only access, and everyone else also has read-only access. The `PosixPermissions` wrapper class has methods for converting between these concise strings and a `Set` of individual `PosixPermission` enum constants. The enum constants are the nine combinations of `OWNER`, `GROUP`, and `OTHERS` with `READ`, `WRITE`, and `EXECUTE`. Here is a JShell example showing these file permission conversion routines:

```
jshell> Set<PosixFilePermission> perms =
  PosixFilePermissions.fromString("rwxr-xr--");
perms ==> [OWNER_READ, OWNER_WRITE, OWNER_EXECUTE, GROUP_READ,
  GROUP_EXECUTE, OTHERS_READ]

jshell> Set<PosixFilePermission> nPerms =
  Set.of(PosixFilePermission.OWNER_READ, PosixFilePermission.GROUP_READ);
nPerms ==> [GROUP_READ, OWNER_READ]

jshell> PosixFilePermissions.toString(nPerms)
$7 ==> "r--r-----"
```

2 The Institute of Electrical and Electronics Engineers, a nonprofit standards organization.

You can further convert the `Set<PosixFilePermission>` into a `FileAttribute` to be used with the `Files` class `createFile()` or `createDirectory()` operations:

```
jshell> PosixFilePermissions.asFileAttribute(nPerms)
$8 ==> java.nio.file.attribute.PosixFilePermissions$1@ed17bee

jshell> $8.name()
$9 ==> "posix:permissions"

jshell> $8.value()
$10 ==> [OWNER_READ, GROUP_READ]

jshell> Files.createFile(Path.of("/tmp/xx"), $8);
$41 ==> /tmp/xx

jshell> /exit

$ ls -l /tmp/xx
-r--r-----  1 ian  wheel  0 Dec 23 11:14 /tmp/xx
$
```

We see the file was created with only owner-read and group-read permissions, as requested. Note that on *nix systems there is a user setting `umask` that may remove or mask out permissions, so what you ask for may not be exactly what you get.

You can examine the attributes of a file using the `FileAttribute` interface or its filesystem-specific subtypes. Here we'll use the `PosixFileAttributeView` to show the owner and permissions of the file we created:

```
PosixFileAttributes attrs =
  Files.getFileAttributeView(filePath,
    PosixFileAttributeView.class)
  .readAttributes();
System.out.format("File %s Owned by %s has perms %s%n",
  filePath,
  attrs.owner().getName(),
  PosixFilePermissions.toString(attrs.permissions()));
```

There are other filesystem-specific views, such as `DosFileAttributeView` for use on FAT (file allocation table) filesystems. FAT was copied from CPM-86 into the earliest releases of MS-DOS, and expanded versions of it are still in use on USB memory cards and in consumer devices.

See Also

There is official documentation for `Files` and `Path`.

10.3 Creating and Deleting Files or Directories

Problem

You need to create a new file on disk but not immediately write any data into it. You need to create a directory before you can create files in it. Or you may need to delete a file or directory.

Solution

To create a file, use `Files.createFile()`. To create a directory, use `Files.createDirectory()` or `Files.createDirectories()`. To delete, use the `Files` object's `delete(Path)` or `deleteIfExists(Path)` method. These methods delete the files referred to by the `Path` argument (subject, of course, to permissions) and directories (subject to permissions and to the directory being empty).

Discussion

For an empty file, use a `java.nio.file.Files` object's `createFile(Path)` method. Use the `Files` class's `createDirectory()` or `createDirectories()` method to create a directory.

Files

You could easily create a new file by constructing a `FileOutputStream` or `FileWriter` (see Recipe 14.6). But then you'd have to remember to close it as well. Sometimes you want a file to exist, but you don't want to bother putting anything into it. This might be used, for example, as a simple form of interprogram communication: one program could test for the presence of a file and interpret that to mean that the other program has reached a certain state. An alternative simple semaphore is to use the `CREATE_NEW` option, which will fail if the file exists. Example 10-5 is code that simply creates an empty file for each name you give.

Example 10-5. main/src/main/java/dir_file/Creat.java (creation of a file on disk)

```
/** Create file(s) by name. Final "e" omitted in homage to UNIX system call. */
public class Creat {
  public static void main(String[] argv) throws IOException {

    // Ensure that a filename (or something) was given in argv[0]
    if (argv.length == 0) {
      throw new IllegalArgumentException("Usage: Creat filename [...]");
    }

    for (String arg : argv) {
      // Constructing a Path object doesn't affect the disk, but
```

```
      // the Files.createFile() method does.
      final Path p = Path.of(arg);
      final Path created = Files.createFile(p);
      System.out.println(created);
    }
  }
}
```

`java.nio.file.createFile()` has an overload that takes a second argument of type `OpenOption`. This is an empty interface that is implemented by the `StandardOpenOption` enumeration. These options are listed in Table 10-5.

Directories

Of the two methods used for creating directories, `createDirectory()` creates just one directory, whereas `createDirectories()` creates any intermediate directories that are needed. For Unix/Linux folks, this is the difference between `mkdir` and `mkdir -p`. For example, if */home/ian* exists and is a directory, the call:

```
shell> Files.createDirectory(Path.of("/Users/ian/abc"))
$11 ==> /Users/ian/abc
```

will succeed (unless the directory is already there), but the call:

```
jshell> Files.createDirectory(Path.of("/Users/ian/once/twice/again"))
```

will fail with a `java.nio.file.NoSuchFileException` because the directory named *once* does not exist. To create this path of directories, as you might expect by now, use the plurally named `createDirectories()`:

```
jshell> Files.createDirectories(Path.of("/Users/ian/once/twice/again"))
$14 ==> /Users/ian/once/twice/again
```

Both variants return a `Path` object referring to the new directory if they succeed, and throw an exception if they fail. Notice that it is possible (but not probable) for `createDirectories()` to create some of the directories and then fail; in this case, the newly created directories are left in the filesystem.

Deletion is not complicated. Simply construct a `Path` object for the file you wish to delete, and call the static `Files.delete()` method:

```
public class Delete {
  public static void main(String[] argv) throws IOException {

    String fileToDelete = "Delete.java~";
    System.out.println("Deleting " + fileToDelete);
    // Construct a File object for the backup created by editing
    // this source file. The file probably already exists.
    // Some text editors create backups by putting ~ at end of filename.
    Path bkup = Path.of(fileToDelete);
    // Now, delete it:
```

```
    if (Files.exists(bkup)) {
      Files.delete(bkup);
      System.out.println("Done");
    } else {
      System.out.println("File not found");
    }
  }
}
```

Recall the caveat about permissions in the introduction to this chapter: if you don't have permission, you can get a return value of false or, possibly, an `Exception`. Note also that there are some differences between platforms. Some versions of Windows allow Java to remove a read-only file, but Unix does not allow you to remove a file unless you have write permission on the directory it's in. Nor does Unix allow you to remove a directory that isn't empty (there is even an exception, `DirectoryNotEmptyException`, for the latter case). Here is a version of `Delete` with reporting of success or failure:

```
public class Delete2 {

  static boolean hard = false; // True for delete, false for deleteIfExists

  public static void main(String[] argv) {
    for (String arg : argv) {
      if ("-h".equals(arg)) {
        hard = true;
        continue;
      }
      delete(arg);
    }
  }

  public static void delete(String fileName) {
    // Construct a File object for the file to be deleted.
    final Path target = Path.of(fileName);

    // Now, delete it:
    if (hard) {
      try {
        System.out.print("Using Files.delete(): ");
        Files.delete(target);
        System.err.println("** Deleted " + fileName + " **");
      } catch (IOException e) {
        System.out.println("Deleting " + fileName + " threw " + e);
      }
    } else {
      try {
        System.out.print("Using deleteIfExists(): ");
        if (Files.deleteIfExists(target)) {
          System.out.println("** Deleted " + fileName + " **");
        } else {
```

```
            System.out.println(
              "Deleting " + fileName + " returned false.");
          }
        } catch (IOException e) {
          System.out.println("Deleting " + fileName + " threw " + e);
        }
      }
    }
}
```

The `-h` option allows this program to switch between `delete()` and `deleteIfExists()`; you can see the difference by running it on things that exist, don't exist, and are not empty, using both methods. The output looks something like this on my Unix box:

```
$ ls -ld ?
-rw-r--r--  1 ian  512   0 Dec 21 16:35 a
drwxr-xr-x  2 ian  512  64 Dec 21 16:35 b
drwxr-xr-x  3 ian  512  96 Dec 21 16:22 c
$ java -cp target/classes dir_file.Delete2 a b c d
Using deleteIfExists(): ** Deleted a **
Using deleteIfExists(): ** Deleted b **
Using deleteIfExists(): Deleting c threw
  java.nio.file.DirectoryNotEmptyException: c
Using deleteIfExists(): Deleting d returned false.

# Here I put the files back the way they were, then run again with -h
$ java -cp target/classes dir_file.Delete2 -h a b c d
Using Files.delete(): ** Deleted a **
Using Files.delete(): ** Deleted b **
Using Files.delete(): Deleting c threw
  java.nio.file.DirectoryNotEmptyException: c
Using Files.delete(): Deleting d threw java.nio.file.NoSuchFileException: d
$ ls -l c
total 2
drwxr-xr-x  2 ian  ian  512 Oct  8 16:50 d
$ java dir_file.Delete2 c/d c
Using deleteIfExists(): ** Deleted c/d **
Using deleteIfExists(): ** Deleted c **
$
```

10.4 Changing a File's Name or Other Attributes

Problem

You need to change a file's name on disk or some of its other attributes, such as setting the file to read-only or changing its modification time.

Solution

To change the name (or location), use a `java.nio.file.Files` static `move()` method. For other attributes, use `setLastModifiedTime()` to change the timestamp, or use one of several other setters for mode or permission attributes.

Discussion

Similar to the Unix command line, there is no separate rename operation; the move methods provide all functions for putting a file somewhere else, whether that is under the same name in a different directory, a different name in the same directory, or a different name on a different disk or filesystem. Accordingly, the `Files.move()` method requires two `Path` objects, one referring to the existing file and another referring to the new name. Then call the `Files.move()` method, passing both `Path` objects, first the existing file and then the desired name. This is easier to see than to explain, so here goes:

```
public class Rename {
  public static void main(String[] argv) throws IOException {

    // Construct the Path object. Does NOT create a file on disk!
    final Path p = Path.of("MyCoolDocument"); // The file we will rename

    // Setup for the demo: create a new "old" file
    final Path oldName = Files.exists(p) ? p : Files.createFile(p);

    // Rename the file to "mydoc.bak"
    // Renaming needs a Path object for the target.
    final Path newName = Path.of("Mydoc.bak");
    Files.deleteIfExists(newName); // In case previous run left it there
    Path p2 = Files.move(oldName, newName);
    System.out.println(p + " renamed to " + p2);

    // What if a directory exists and you try to move a file into it?
    Path target = Path.of("temp");
    if (!Files.isDirectory(target)) {
      Files.createDirectory(target);
    }
    try {
      Files.move(p2, target);
    } catch (java.nio.file.FileAlreadyExistsException ex) {
      System.out.println("Caught expected FileAlreadyExistsException");
    }
    // Cleanup time!
    Files.delete(p2);
    Files.delete(target);
  }
}
```

For changing the attributes, there are several methods available, listed in Table 10-6. Each of these has a return value of type `boolean`, with `true` meaning success.

Table 10-6. Legacy file attribute setters

Method signature	Description
`setExecutable(boolean executable)`	Convenience method to set owner's execute permission for this file
`setExecutable(boolean executable, boolean ownerOnly)`	Sets the owner's or everybody's execute permission for this file
`setLastModified(long time)`	Sets the last-modified time of the file or directory that this file names
`setReadable(boolean readable)`	Convenience method to set owner's read permission for this file
`setReadable(boolean readable, boolean ownerOnly)`	Sets the owner's or everybody's read permission for this file
`setReadOnly()`	Convenience method for `setReadable(false)`
`setWritable(boolean writable)`	A convenience method to set the owner's write permission for this file
`setWritable(boolean writable, boolean ownerOnly)`	Sets owner's or everybody's write permission for this file

For the methods that take two arguments, the first enables or disables the feature on the given file that matches the method name, and the second controls whether the operation applies to the owner only or to everyone. The second argument is ignored if the file lives on a filesystem that doesn't support multiuser permissions or if the operating system doesn't support that. All the methods described in this recipe return `true` if they succeed and `false` otherwise.

For example, `boolean setReadable(boolean readable, boolean ownerOnly)` lets you specify who can read the given file. The `readable` argument is true or false depending on whether you want it to be readable or not. A value of `false` for the `ownerOnly` argument tries to extend the readability choice to all users on a multiuser operating system, and is ignored if not applicable.

`setLastModified()` allows you to play games with the modification time of a file. This is normally not a good game to play, but it is useful in some types of backup/restore programs. This method takes an argument that is the number of milliseconds (not seconds) since the beginning of Unix time (January 1, 1970). You can get the original value for the file by calling `getLastModified()` (see Recipe 10.2), or you can get the value for a given date by calling the `ZonedDateTime`'s `toInstant().getEpochSecond()` method (see Recipe 6.3) and multiplying by 1,000 to convert seconds to milliseconds.

I encourage you to explore the operation of these methods using JShell (see Recipe 1.5). I'd suggest having a second window in which you can run `ls -l` or `dir` commands to see how the file is affected. Example 10-6 shows some of these methods on the legacy `File` class being explored in JShell.

Example 10-6. Exploring files

```
jshell> var f = File.createTempFile("foo", "bar");
f ==> /tmp/foo9391300789087780984bar

jshell> f.createNewFile();
$4 ==> false

jshell> f.setReadOnly();
$5 ==> true

jshell> f.canRead();
$6 ==> true

jshell> f.canWrite();
$7 ==> false

jshell> f.setReadable(true);
$8 ==> true

jshell> f.canWrite();
$9 ==> false

jshell> f.setReadable(false, false);
$10 ==> true

jshell> f.canWrite();
$11 ==> false
```

10.5 About InputStreams/OutputStreams and Readers/Writers

Problem

You need to read or write a file with text or binary data.

Solution

Use a `Reader` or `Writer` for text, or an `InputStream` or `OutputStream` for binary data.

Discussion

Java provides two sets of classes for reading and writing. The `InputStream/OutputStream` section of package java.io (see Figure 10-1) is for reading or writing bytes of data. Older languages tended to assume that a byte (which is a machine-specific collection of bits, usually eight bits on modern computers) is exactly the same thing as a character—a letter, digit, or other linguistic element. However, Java is designed to be used internationally, and eight bits is simply not enough to handle the many different character sets used around the world. Script-based languages, and pictographic languages like Chinese and Japanese, each have many more than 256 characters, the maximum that can be represented in an 8-bit byte. The unification of these many character code sets is called, not surprisingly, Unicode. Both Java and XML use Unicode as their character sets, allowing you to read and write text in any of these human languages. You should use `Readers` and `Writers`, not `Streams`, for textual data.

Unicode itself doesn't solve the entire problem. Many of these human languages were used on computers long before Unicode was invented, and they didn't all pick the same representation as Unicode. And they all have zillions of files encoded in a particular representation that isn't Unicode. So routines are needed when reading and writing to convert between Unicode `String` objects used inside the Java machine and the particular external representation in which a user's files are written. These converters are packaged inside a powerful set of classes called `Readers` and `Writers`. `Readers` and `Writers` should always be used instead of `InputStreams` and `OutputStreams` when you want to deal with characters instead of bytes. We'll see more on this conversion, and how to specify which conversion, a little later in this chapter.

See Also

One topic *not* addressed in depth here is the reading/writing capabilities of `Channels` classes in the Java new I/O (NIO) package.[3] This part of NIO is more complex to use than either `Files` or the input/output streams, and the benefits accrue primarily in large-scale server-side processing. Recipe 4.6 provides one example of using NIO. The NIO package is given full coverage in the book *Java NIO* by Ron Hitchens (O'Reilly).

Another topic not covered here is that of having the read or write occur concurrently with other program activity. This requires the use of threads, or multiple flows of control within a single program. Threaded I/O is a necessity in many programs: those reading from slow devices such as tape drives, those reading from or writing to net-

3 A poor choice of name: it was new in Java SE 1.4. But it is newer than `InputStream/OutputStream` (Java 1.0) and `Readers/Writers` (1.1).

work connections, and those with a GUI. For this reason, the topic is given considerable attention, in the context of multithreaded applications, in Chapter 11.

For traditional I/O topics, Elliotte Rusty Harold's *Java I/O*, although somewhat dated, should be considered the antepenultimate documentation. The penultimate reference is the Javadoc documentation, while the ultimate reference is, if you really need it, the source code for the Java API. Due in part to the quality of the Javadoc documentation, I have not needed to refer to the source code in writing this chapter.

10.6 Reading and Writing Files

Problem

The Java documentation doesn't have methods for opening files. How do I open a text file and then either read it a line at a time, or get a collection of all the lines?

Solution

Use the `Files::lines()` method, which returns a `Stream` of `Strings`. Or, use `Files.readAllLines()`, which returns a `List<String>`. Or, use `Files.newBufferedReader()`, `Files.newBufferedWriter()`, `Files.newInputStream()`, and `Files.newOutputStream()`. Or, construct a `FileReader`, a `FileInputStream`, or a `BufferedReader`, and use the older `while ((line == readLine()) != null)` pattern.

Discussion

There is no explicit open operation,[4] perhaps as a kind of rhetorical flourish of the Java API's object-oriented design.

The quickest way to process a text file a line at a time is to use `Files.lines()`, which takes a `Path` argument and returns a functional `Stream<String>` into which it feeds the lines from the file:

```
Files.lines(Path.of("myFile.txt")).forEach(System.out::println);
```

The `Files` class has several other static methods that open a file and read some or all of it:

`List<String> Files.readAllLines(Path)`
: Reads the whole file into a `List<String>`

`String Files.readString(Path)`
: Returns the whole file as a multi-line string

4 Not strictly true; there is, but only in the `java.nio.FileChannel` class, which we're not covering.

`byte[] Files.readAllBytes`
: Reads the whole file into an array of bytes

There is a series of methods with names like `newReader()`, `newBufferedWriter()`, etc., each of which takes a `Path` argument and returns the appropriate `Reader/Writer` or `InputStream/OutputStream`. As discussed in Recipe 10.2, a `Path` is a descriptor for an abstract path (filename) that may or may not exist. The explicit constructors for a `FileReader`, `FileWriter`, `FileInputStream`, or `FileOutputStream` take a filename or an instance of the older `File` class containing the path. These operations correspond to the "open" operation in most other languages' I/O packages.

Historically, Java used to require use of the code pattern `while ((line == readLine()) != null)` to read lines from a `BufferedReader`. This still works, of course, and will continue to work until the last JavaBean sets in the west, in the far distant future.

Example 10-7 shows the code for each of these ways to read lines from a file.

Example 10-7. main/src/main/java/io/ReadLines.java (reading lines from a file)

```
System.out.println("Using Path.lines()");
Files.lines(Path.of(fileName)).forEach(System.out::println);

System.out.println("Using Path.readAllLines()");
List<String> lines = Files.readAllLines(Path.of(fileName));
lines.forEach(System.out::println);

System.out.println("Using BufferedReader.lines().forEach()");
new BufferedReader(new FileReader(fileName)).lines().forEach(s -> {
  System.out.println(s);
});

System.out.println("The old-fashioned way");
BufferedReader is = new BufferedReader(new FileReader(fileName));
String line;
while ((line = is.readLine()) != null) {
  System.out.println(line);
}
```

The `BufferedReader` construction shown here illustrates what is commonly but informally called *constructor chaining*; the `FileReader` constructed by the "inner" call is passed directly to the `BufferedReader` constructor. This is also an example of the GoF Decorator pattern, where one class adds functionality—in this case, line reading—to another.

Most of these methods can throw the checked exception `IOException`, so you must have a `throws` clause or a `try/catch` around these invocations.

If you create an `InputStream`, `OutputStream`, `Reader`, or `Writer`, you should close it when finished. This avoids memory leaks and, in the case of writing, ensures that all buffered data is actually written to disk. One way to ensure this is not forgotten is to use the `try-with-resources` syntax. This puts the declaration and definition of a `Closeable` resource into the `try` statement:

```
static void oldWayShorter() throws IOException {
  try (BufferedReader is =
    new BufferedReader(new FileReader(INPUT_FILE_NAME));
    BufferedOutputStream bytesOut = new BufferedOutputStream(
      new FileOutputStream(OUTPUT_FILE_NAME.replace("\\.", "-1."));) {

    // Read from is, write to bytesOut
    String line;
    while ((line = is.readLine()) != null) {
      line = doSomeProcessingOn(line);
      bytesOut.write(line.getBytes("UTF-8"));
      bytesOut.write('\n');
    }

  }
}
```

The `lines()` and read-related methods in `Files` obviate the need for closing the resource, but not the need for handling `IOException`; the compiler or IDE will remind you if you forget those.

There are options that can be passed to the `Files` methods that open a file; these are discussed in the sidebar "Understanding I/O Options: StandardOpenOptions, FileAttribute, PosixFileAttribute, and More" on page 370.

To read the entire contents of a file into a single string in Java 8+, use `Files.readString()`:

```
String input = Files.readString(Path.of(INPUT_FILE_NAME));
```

In older Java versions, use my `FileIO.readerToString()` method. Either of these will read the entire named file into one long string, with embedded newline (`\n`) characters between each line. To read a binary file, use `Files.readAllBytes()` instead.

When you need to read or write binary data, as opposed to text, just use a `DataInputStream` or `DataOutputStream`.

The `Stream` classes have been in Java since the beginning of time and are optimal for reading and writing bytes rather than characters. The data layer over them, comprising `DataInputStream` and `DataOutputStream`, is configured for reading and writing binary values, including all of Java's built-in types. Suppose that you want to write a

binary integer plus a binary floating-point value into a file and read it back later. This code shows the writing part:

```
public class WriteBinary {
  public static void main(String[] argv) throws IOException {
    int i = 42;
    double d = Math.PI;
    String FILENAME = "binary.dat";
    DataOutputStream os = new DataOutputStream(
      new FileOutputStream(FILENAME));
    os.writeInt(i);
    os.writeDouble(d);
    os.close();
    System.out.println("Wrote " + i + ", " + d + " to file " + FILENAME);
  }
}
```

Should you need to write all the fields from an object, you should probably use one of the methods described in Recipe 14.6.

Formatted Printing

Should you want an easy way to use the `java.util.Formatter` class's capability for simple printing tasks, use the `printf()` methods in `PrintStream`/`PrintWriter`. The `Formatter` class is patterned after C's `printf` routines and has been discussed in Recipe 3.2.

10.7 Scanning Input with StreamTokenizer, Scanner, Parsers

Problem

You need to scan a file with more fine-grained resolution than the `readLine()` method of the `BufferedReader` class and its subclasses.

Solution

Use a `StreamTokenizer`, `readLine()` and a `StringTokenizer`, the `Scanner` class, regular expressions (Chapter 4), or one of several third-party parser generators.

Discussion

Though you could, in theory, read a file one character at a time and analyze each character, that is a pretty low-level approach. The `read()` method in the `Reader` class is defined to return `int` so that it can use the time-honored value `-1` (defined as EOF in Unix `<stdio.h>` for decades) to indicate that you have read to the end of the file:

main/src/main/java/io/ReadCharsOneAtATime.java

```
public class ReadCharsOneAtATime {

    void doFile(Reader is) throws IOException {
        int c;
        while ((c=is.read( )) != -1) {
            System.out.print((char)c);
        }
    }
}
```

Notice the cast to `char`; the program compiles fine without it, but it does not print correctly because `c` is declared as `int`. Variable `c` must be declared `int` to be able to compare against the end-of-file value `-1`. For example, the integer value corresponding to capital A treated as an `int` prints as 65, whereas with `(char)` it prints the character `A`.

We discussed the `StringTokenizer` class extensively in Recipe 3.1. The combination of `readLine()` and `StringTokenizer` provides a simple means of scanning a file. Suppose you need to read a file in which each line consists of a name like *user@host.domain*, and you want to split the lines into users and host addresses. You could use this:

```
public class ScanStringTok {

  public static void main(String[] av) throws IOException {
    if (av.length == 0)
      System.err.printf("Usage: %s filename [...]%n",
        ScanStringTok.class.getSimpleName());
    else
      for (int i=0; i<av.length; i++)
        process(av[i]);
  }

  static void process(String fileName) {
    String s = null;
    try (BufferedReader is =
        new BufferedReader(new FileReader(fileName));) {
        while ((s = is.readLine()) != null) {
          StringTokenizer st = new StringTokenizer(s, "@", true);
          String user = (String)st.nextElement();
          st.nextElement();
          String host = (String)st.nextElement();
          System.out.println("User name: " + user +
            "; host part: " + host);

          // Do something useful with the user and host parts...
        }
    } catch (NoSuchElementException ix) {
      System.err.println("Malformed input " + s);
```

```
        } catch (IOException e) {
          System.err.println(e);
        }
      }
    }
```

The `StreamTokenizer` class in java.util provides slightly more capabilities for scanning a file. It reads characters and assembles them into words, or tokens. It returns these tokens to you along with a type code describing the kind of token it found. This type code is one of four predefined types (`StringTokenizer.TT_WORD`, `TT_NUMBER`, `TT_EOF`, or `TT_EOL` for the end-of-line) or the `char` value of an ordinary character (such as 32 for the space character). Methods such as `ordinaryCharacter()` allow you to specify how to categorize characters, while others, such as `{asciidoctor-version}slashSlashComment()`, allow you to enable or disable features.

Example 10-8 shows a `StreamTokenizer` used to implement a simple immediate-mode stack-based calculator:

```
2 2 + =
4
22 7 / =
3.141592857
```

The program reads tokens as they arrive from the `StreamTokenizer`. Numbers are put on the stack. The four operators (`+`, `-`, `*`, and `/`) are immediately performed on the two elements at the top of the stack, and the result is put back on the top of the stack. The `=` operator causes the top element to be printed, but is left on the stack so that you can chain operations, as in this example:

```
4 5 * = 2 / =
20.0
10.0
```

Example 10-8. main/src/main/java/io/SimpleCalcStreamTok.java (simple calculator using `StreamTokenizer`)

```
public class SimpleCalcStreamTok {
  /** The StreamTokenizer Input */
  protected  StreamTokenizer tf;
  /** The Output File */
  protected PrintWriter out = new PrintWriter(System.out, true);
  /** The operand stack */
  protected Stack<Double> s = new Stack<>();
  /** The name of a variable */
  protected String variable;

  protected void doCalc() throws IOException {
    int iType;
    double tmp;
```

```
    while ((iType = tf.nextToken()) != StreamTokenizer.TT_EOF) {
      switch(iType) {
      case StreamTokenizer.TT_NUMBER: // Found a number, push value to stack
        push(tf.nval);
        break;
      case StreamTokenizer.TT_WORD:
        // Found a variable, save its name. Not used here.
        variable = tf.sval;
        break;
      case '+':
        // + operator is commutative.
        push(pop() + pop());
        break;
      case '-':
        // - operator: order matters.
        tmp = pop();
        push(pop() - tmp);
        break;
      case '*':
        // Multiply is commutative
        push(pop() * pop());
        break;
      case '/':
        // Handle division carefully: order matters!
        tmp = pop();
        push(pop() / tmp);
        break;
      case '=':
        out.println(peek());
        break;
      default:
        out.println("What's this? iType = " + iType);
      }
    }
  }
  void push(double val) {
    s.push(Double.valueOf(val));
  }
  double pop() {
    return ((Double)s.pop()).doubleValue();
  }
  double peek() {
    return ((Double)s.peek()).doubleValue();
  }
  void clearStack() {
    s.removeAllElements();
  }
```

The `Scanner` class is useful when you want to scan a text file consisting of various numbers and strings in a known format. You can read values of these types with the `Scanner`'s `next()` methods.

The `Scanner` class lets you read an input source by tokens, somewhat analogous to the `StreamTokenizer` described in Recipe 10.7. The `Scanner` is more flexible in some ways (it lets you break tokens based on spaces or regular expressions) but less flexible in others (you need to know the kind of token you are reading). This class bears some resemblance to the C-language `scanf()` function, but in the `Scanner` you specify the input token types by calling methods like `nextInt()`, `nextDouble()`, and so on. Here is a simple example of scanning:

```
String sampleDate = "25 Dec 1988";

try (Scanner sDate = new Scanner(sampleDate)) {
  int dayOfMonth = sDate.nextInt();
  String month = sDate.next();
  int year = sDate.nextInt();
  System.out.printf("%d-%s-%02d%n", year, month, dayOfMonth);
}
```

The `Scanner` recognizes Java's eight built-in types, in addition to `BigInteger` and `BigDecimal`. It can also return input tokens as `String`s or by matching regular expressions (see Chapter 4). Table 10-7 lists the "next" methods and corresponding "has" methods; the "has" method returns true if the corresponding "next" method would succeed. There is no `nextString()` method; either use `nextLine()` to read an entire line as a string, or use `next()` to get the next token as a `String`.

Table 10-7. Scanner methods

Returned type	has method	next method	Comment
`String`	`hasNext()`	`next()`	The next complete token from this scanner
`String`	`hasNext(Pattern)`	`next(Pattern)`	The next string that matches the given regular expression (regex)
`String`	`hasNext(String)`	`next(String)`	The next token that matches the regex pattern constructed from the specified string
`BigDecimal`	`hasNextBigDecimal()`	`nextBigDecimal()`	The next token of the input as a `BigDecimal`
`BigInteger`	`hasNextBigInteger()`	`nextBigInteger()`	The next token of the input as a `BigInteger`
`boolean`	`hasNextBoolean()`	`nextBoolean()`	The next token of the input as a `boolean`
`byte`	`hasNextByte()`	`nextByte()`	The next token of the input as a `byte`
`double`	`hasNextDouble()`	`nextDouble()`	The next token of the input as a `double`
`float`	`hasNextFloat()`	`nextFloat()`	The next token of the input as a `float`
`int`	`hasNextInt()`	`nextInt()`	The next token of the input as an `int`
`String`	N/A	`nextLine()`	Reads up to the end-of-line, including the line ending
`long`	`hasNextLong()`	`nextLong()`	The next token of the input as a `long`
`short`	`hasNextShort()`	`nextShort()`	The next token of the input as a `short`

The `Scanner` class is constructed with an input source, which can be an `InputStream`, a `String`, or `Readable` (`Readable` is an interface that `Reader` and all its subclasses implement).

One way to use the `Scanner` class is based on the Iterator pattern, using `while (scanner.hasNext())` to control the iteration. Example 10-9 shows the simple calculator from Recipe 10.7 rewritten[5] to use the `Scanner` class.

Example 10-9. main/src/main/java/io/simpleCalcScanner.java (simple calculator using `java.util.Scanner`)

```
public class SimpleCalcScanner {
  /** The Scanner */
  protected Scanner scan;

  /** The output */
  protected PrintWriter out = new PrintWriter(System.out, true);

  /** The variable name (not used in this version) */
  protected String variable;

  /** The operand stack; no operators are pushed, so it can be a stack of Double */
  protected Stack<Double> s = new Stack<>();

  /* Driver - main program */
  public static void main(String[] args) throws IOException {
    if (args.length == 0)
      new SimpleCalcScanner(
        new InputStreamReader(System.in)).doCalc();
    else
      for (String arg : args) {
        new SimpleCalcScanner(arg).doCalc();
      }
  }

  /** Construct a SimpleCalcScanner by name */
  public SimpleCalcScanner(String fileName) throws IOException {
    this(new FileReader(fileName));
  }

  /** Construct a SimpleCalcScanner from an open Reader */
  public SimpleCalcScanner(Reader rdr) throws IOException {
    scan = new Scanner(rdr);
  }
```

5 If this were code in a maintained project, I might factor out some of the common code among these two calculators, as well as the one in Recipe 5.13, and divide the code better using interfaces. However, this would detract from the simplicity of self-contained examples.

```
/** Construct a SimpleCalcScanner from a Reader and a PrintWriter */
public SimpleCalcScanner(Reader rdr, PrintWriter pw) throws IOException {
  this(rdr);
  setWriter(pw);
}

/** Change the output to go to a new PrintWriter */
public void setWriter(PrintWriter pw) {
  out = pw;
}

protected void doCalc() throws IOException {
  double tmp;

  while (scan.hasNext()) {
    if (scan.hasNextDouble()) {
      push(scan.nextDouble());
    } else {
      String token;
      switch(token = scan.next()) {
      case "+":
        // Found + operator, perform it immediately.
        push(pop() + pop());
        break;
      case "-":
        // Found - operator, perform it (order matters).
        tmp = pop();
        push(pop() - tmp);
        break;
      case "*":
        // Multiply is commutative
        push(pop() * pop());
        break;
      case "/":
        // Handle division carefully: order matters!
        tmp = pop();
        push(pop() / tmp);
        break;
      case "=":
        out.println(peek());
        break;
      default:
        out.println("What's this? " + token);
        break;
      }
    }
  }
}

void push(double val) {
  s.push(Double.valueOf(val));
}
```

```
  double pop() {
    return ((Double)s.pop()).doubleValue();
  }

  double peek() {
    return ((Double)s.peek()).doubleValue();
  }

  void clearStack() {
    s.removeAllElements();
  }
}
```

Although the `StreamTokenizer` class (see Recipe 10.7) and `Scanner` (see Recipe 10.7) are useful, they know only a limited number of tokens and have no way of specifying that the tokens must appear in a particular order. To do more advanced scanning, particularly if you need to parse a file whose structure can be described as grammatical (in the sense of computer languages, not natural languages), you should consider using a tool known as a *parser generator*.

Parser generators have a long history in computer science. The best-known examples are the C-language `yacc` (Yet Another Compiler Compiler) and `lex`, released with Seventh Edition Unix in the 1970s and discussed in *lex & yacc* by Doug Brown et al. (O'Reilly), and their open source clones *bison* and *flex*. These tools let you specify the lexical structure of your input using some pattern language such as regular expressions (see Chapter 4). For example, you might say that an email address consists of a series of alphanumerics, followed by an at sign (@), followed by a series of alphanumerics with periods embedded, like this:

```
email:  [A-Za-z0-9]+@[A-Za-z0-0.]+
or
email: \w+@[\w.]+
```

The tool then writes code that recognizes the characters you have described. These tools also have a grammatical specification, which says, for example, that the keyword `EMAIL` must appear, followed by a colon, followed by a `name` token, as previously defined.

There are several good third-party parser generator tools for Java. They vary widely based on complexity, power, and ease of use:

- One of the best known and most elaborate is ANTLR.
- JavaCC is an open source project.
- JParsec lets you write the parser in straight Java, so it's all built at compile time (most of the others require a separate parse generation step, with the build and debugging issues that raises). JParsec is on GitHub.

- JFlex and CUP work together like the original *yacc* and *lex*, as grammar parser and lexical scanner, respectively.
- Parboiled uses *Parsing Expression Grammar* (PEG) to also build the parser at compile time. See GitHub for more information.
- The *Rats!* parser generator was part of the eXTensible Compiler Project at New York University. (The project's website is no longer available.)
- There are others—a more complete list was maintained at Java Source; it is still online but is not being updated.

These parser generators can be used to write grammars for a wide variety of programs, from simple calculators—such as the one in Recipe 10.7—through HTML and CORBA/IDL, up to full Java and C/C++ parsers. Examples of these are included with the downloads. Unfortunately, the learning curve for parsers in general precludes providing a simple and comprehensive example here, let alone comparing them intelligently. Refer to the documentation and the numerous examples provided with each distribution.

As an alternative to using one of these parser generators, you could simply roll your own recursive descent parser; and once you learn how to do so, you may find it's not really that difficult, quite possibly even less hassle than dealing with the extra parser generator software (depending on the complexity of the grammar involved, obviously).

Java developers have a range of choices, including simple line-at-a-time scanners using `StringTokenizer`, fancier token-based scanners using `StreamTokenizer`, a `Scanner` class to scan simple tokens (see earlier in Recipe 10.7), regular expressions (see Chapter 4), and third-party solutions including grammar-based scanners based on the parsing tools listed here.

10.8 Reading from the Standard Input or from the Console/Controlling Terminal

Problem

You want to read from the program's standard input or directly from the program's controlling terminal or console terminal.

Solution

For the standard input, read bytes by wrapping a `BufferedInputStream()` around `System.in`. For reading text, use an `InputStreamReader` and a `BufferedReader`. For the console or controlling terminal, use Java's `System.console()` method to obtain a `Console` object, and use its methods.

Discussion

Sometimes you really do need to read from the standard input, or console. One reason for this is that simple test programs are often console-driven. Another is that some programs naturally require a lot of interaction with the user and you want something faster than a GUI (consider an interactive mathematics or statistical exploration program). Yet another is piping the output of one program directly to the input of another, a very common operation among Unix users and quite valuable on other platforms, such as Windows, that support this operation.

Standard input and output streams

All desktop operating systems support the notion of *standard input* (a keyboard, a file, or the output from another program) and *standard output* (a terminal window, a printer, a file on disk, or the input to yet another program).[6] Most such systems also support a standard error output so that error messages can be seen by the user even if the standard output is being redirected. When programs on these platforms are started up, the system assigns the three streams to particular platform-dependent handles, or *file descriptors*. The net result is that ordinary programs on these operating systems can read the standard input or write to the standard output or standard error stream without having to open any files or make any other special arrangements.

Java continues this tradition and enshrines it in the `System` class. The static variables `System.in`, `System.out`, and `System.err` are connected to the three operating system streams before your program begins execution (an application is free to reassign these; see Recipe 10.11). So, to read the standard input, you need only refer to the variable `System.in` and call its methods. For example, to read one byte from the standard input, you call the read method of `System.in`, which returns the byte in an `int` variable:

```
int b = System.in.read( );
```

6 Even IBM's MVT and MVS operating systems provided `SYSIN` and `SYSOUT` in their batch Job Control Language.

But is that enough? No, because the `read()` method can throw an `IOException`. So you must either declare that your program throws an `IOException`:

```
public static void main(String args[]) throws IOException {
...
}
```

or you can put a `try`/`catch` block around the `read()` method:

```
int b = 0;
try {
  b = System.in.read();
  System.out.println("Read this data: " + (char)b);
} catch (Exception e) {
  System.out.println("Caught " + e);
}
```

In this case, it makes sense to print the results inside the `try` block because there's no point in trying to print the value you read, if the `read()` threw an `IOException`.

That code works and gives you the ability to read a byte at a time from the standard input. But most applications are designed in terms of larger units, such as integers, or lines of text. To read a value of a known type, such as `int` or `String`, from the standard input, you can use the `Scanner` class (covered in more detail earlier in Recipe 10.7):

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
String s = sc.nextLine();
```

But this is fairly limiting. To read characters of text with an input character converter so that your program will work with multiple input encodings around the world, use a `Reader` class. The particular subclass that allows you to read lines of characters is a `BufferedReader`. But there's a hitch. Remember I mentioned those two categories of input classes, `Streams` and `Readers`? I also said that `System.in` is a `Stream`, and you want a `Reader`. How do you get from a `Stream` to a `Reader`? A crossover class called `InputStreamReader` is tailor-made for this purpose. Just pass your `Stream` (like `System.in`) to the `InputStreamReader` constructor and you get back a `Reader`, which you in turn pass to the `BufferedReader` constructor. The usual idiom for writing this in Java is to nest the constructor calls:

```
BufferedReader is = new BufferedReader(new InputStreamReader(System.in));
```

You can then read lines of text using the `readLine()` method. This method takes no argument and returns a `String` that is made up for you by `readLine()` containing the characters (converted to Unicode) from the next line of text in the file (the newline character(s) are not included). When there are no more lines of text, the literal value `null` is returned:

```
public class CatStdin {

    public static void main(String[] av) {
        try (BufferedReader is =
              new BufferedReader(new InputStreamReader(System.in))) {
            String inputLine;

            while ((inputLine = is.readLine()) != null) {
                System.out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
    }
}
```

To read a single `Integer` from the standard input, either use a `Scanner` or read a line and parse it using `Integer.parseInt()`. To read a series of integers, one per line, you could combine these with a functional style, since the `BufferedReader` has a `lines()` method that produces a `Stream<String>`:

```
public class ReadStdinIntsFunctional {
  private static Stream<Integer> parseIntSafe(String s) {
    try {
      return Stream.of(Integer.parseInt(s));
    } catch (NumberFormatException e) {
      return Stream.empty();
    }
  }

  public static void main(String[] args) throws IOException {
    try (BufferedReader is =
        new BufferedReader(new InputStreamReader(System.in));) {
      is.lines()
        .flatMap(ReadStdinIntsFunctional::parseIntSafe)
        .forEach(System.out::println);
    }
  }
}
```

The Console (Controlling Terminal)

The `Console` class is intended for reading directly from a program's controlling terminal. When you run an application from a *terminal window* or *command prompt window* on most systems, its console and its standard input are both connected to the terminal, by default. However, the standard input can be changed by piping or redirection on most OSes. If you really want to read from wherever the user is sitting, bypassing any indirections, then the `Console` class is usually your friend.

You cannot instantiate `Console` yourself; you must get an instance from the `System` class's `console()` method. You can then call methods such as `readLine()`, which behaves largely like the method of the same name in the `BufferedReader` class used in the previous recipe.

The following code shows an example of prompting for a name and reading it from the console:

main/src/main/java/io/ConsoleRead.java

```
public class ConsoleRead {
  public static void main(String[] args) {
    String name = System.console().readLine("What is your name?");
    System.out.println("Hello, " + name.toUpperCase());
  }
}
```

One complication is that the `System.console()` method can return `null` if the console isn't connected. So production-quality code should always check for `null` before trying to use `Console`.

The `Console` class is quite useful for reading a password without having it echo. This has been a standard facility of command-line applications for decades, as the most obvious way of preventing *shoulder surfing*—somebody looking over your shoulder to see your password. No-echo password reading works in Java: the `Console` class has a `readPassword()` method that takes a `prompt` argument, intended to be used like: `cons.readPassword("Password:")`. This method returns an array of bytes, which can be used directly in some encryption and security APIs, or can easily be converted into a `String`. It is generally advised to overwrite the byte array after use to prevent security leaks when other code can access the stack, although the benefits of this are probably reduced when you've constructed a `String`. There's an example of this in the online code in *io/ReadPassword.java*.

10.9 Copying a File

Problem

You need to copy a file in its entirety.

Solution

Use one of the Java 11 `Files.copy()` methods. If on an older release, use the explicit read and write methods in the `Readers`/`Writers` or `InputStreams`/`OutputStreams`.

Discussion

The `Files` class has several overloads of a `copy` method that makes quick work of this requirement:

```
long copy(InputStream src, Path dest, CopyOption...) throws IOException;
long copy(Path src, OutputStream dest) throws IOException;
// Returns the Path to the target:
Path copy(Path src, Path dest, CopyOption...) throws java.io.IOException;
```

For example:

```
Path p = Paths.get("my_new_file");
InputStream is = // open some file for reading
long newFileSize = Files.copy(is, p);
```

Long ago, Java's I/O facilities did not package a lot of the common operations like copying one file to another or reading a file into a `String`. So back then I wrote my own package of helper methods. Users of older JDK versions may want to use `FileIO` from my utilities package com.darwinsys.util. Here's a simple demo program that uses `FileIO` to copy a source file to a backup file:

main/src/demo/java/io/FileIoDemo.java

```
package com.darwinsys.io;

import java.io.IOException;

public class FileIoDemo {
    public static void main(String[] av) {
        try {
            FileIO.copyFile("FileIO.java", "FileIO.bak");
            FileIO.copyFile("FileIO.class", "FileIO-class.bak");
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

My `copyFile` method takes several forms, depending on whether you have two filenames, a filename and a `PrintWriter`, and so on. The code for `FileIO` itself is not shown here but is online, in the *darwinsys-api* download.

10.10 Reassigning the Standard Streams

Problem

You need to reassign one or more of the standard streams `System.in`, `System.out`, or `System.err`.

Solution

Construct an `InputStream` or `PrintStream` as appropriate, and pass it to the appropriate set method in the `System` class.

Discussion

The ability to reassign these streams corresponds to what Unix (or DOS command line) users think of as *redirection*, or *piping*. This mechanism is commonly used to make a program read from or write to a file without having to explicitly open it and go through every line of code changing the read, write, print, etc. calls to refer to a different stream object. The open operation is performed by the command-line interpreter in Unix or DOS or by the calling class in Java.

Although you could just assign a new `PrintStream` to the variable `System.out`, best practice is to use the defined method to replace it:

```
String LOGFILENAME = "error.log";
System.setErr(new PrintStream(new FileOutputStream(LOGFILENAME)));
System.out.println("Please look for errors in " + LOGFILENAME);
// Now assume this is somebody else's code; you'll see it
//   writing to stderr...
int[] a = new int[5];
a[10] = 0;  // here comes an ArrayIndexOutOfBoundsException
```

The stream you use can be one that you've opened, as here, or one you inherited:

```
System.setErr(System.out);    // merge stderr and stdout to same output stream.
```

It could also be a stream connected to or from another `Process` you've started (see Recipe 18.1), a network socket, or a URL. Anything that gives you a stream can be used.

10.11 Duplicating a Stream as It Is Written

Problem

You want anything written to a stream, such as the standard output `System.out` or the standard error `System.err`, to appear there but *also* be logged in to a file.

Solution

Subclass `PrintStream` and have its `write()` methods write to two streams. Then use `system.setErr()` or `setOut()` to replace the existing standard stream with a `PrintStream` subclass.

Discussion

Some classes are meant to be subclassed. Here we're just subclassing `PrintStream` and adding a bit of functionality: a second `PrintStream`! I wrote a class called `TeePrintStream`, named after the ancient Unix command `tee`. That command allowed you to duplicate, or "tee off" (a term derived from a plumber's pipe tee, not from golf or the local pest), a copy of the data being written on a pipeline between two programs.

The original Unix `tee` command is used like this: the `|` character creates a pipeline in which the standard output of one program becomes the standard input to the next. This often-used example of pipes shows how many users are logged in to a Unix server:

```
who | wc -l
```

This runs the `who` program (which lists who is logged in to the system, one name per line, along with the terminal port and login time) and sends its output, not to the terminal, but rather into the standard input of the word count (`wc`) program. Here, `wc` is being asked to count lines, not words, hence the `-l` option. To `tee` a copy of the intermediate data into a file, you might say:

```
who | tee wholist | wc -l
```

which creates a file *wholist* containing the data. For the curious, the file *wholist* might look something like this:

```
ian      ttyC0    Mar 14 09:59
ben      ttyC3    Mar 14 10:23
ian      ttyp4    Mar 14 13:46  (laptop.darwinsys.com)
```

So both the previous command sequences would print `3` as their output.

`TeePrintStream` is my attempt to capture the spirit of the `tee` command. It can be used like this:

```
System.setErr(new TeePrintStream(System.err, "err.log"));
// ...lots of code that occasionally writes to System.err... Or might.
```

`System.setErr()` is a means of specifying the destination of text printed to `System.err` (there are also `System.setOut()` and `System.setIn()`). This code results in any messages that printed to `System.err` printing to wherever `System.err` was previously directed (normally the terminal, but possibly a text window in an IDE) and to the file *err.log*.

This technique is not limited to the three standard streams. A `TeePrintStream` can be passed to any method that wants a `PrintStream`, or, for that matter, an `OutputStream`. And you can adapt the technique for `BufferedInputStreams`, `PrintWriters`, `BufferedReaders`, and so on.

Example 10-10 shows the source code for `TeePrintStream`.

Example 10-10. main/src/main/java/io/TeePrintStream.java

```
public class TeePrintStream extends PrintStream {
  /** The original/direct print stream */
  protected PrintStream parent;

  /** The filename we are tee-ing too, if known;
   * intended for use in future error reporting.
   */
  protected String fileName;

  /** The name for when the input filename is not known */
  private static final String UNKNOWN_NAME = "(opened Stream)";

  /** Construct a TeePrintStream given an existing PrintStream,
   * an opened OutputStream, and a boolean to control auto-flush.
   * This is the main constructor, to which others delegate via "this".
   */
  public TeePrintStream(PrintStream orig, OutputStream os, boolean flush)
  throws IOException {
    super(os, true);
    fileName = UNKNOWN_NAME;
    parent = orig;
  }

  /** Construct a TeePrintStream given an existing PrintStream and
   * an opened OutputStream.
   */
  public TeePrintStream(PrintStream orig, OutputStream os)
  throws IOException {
    this(orig, os, true);
  }

  /* Construct a TeePrintStream given an existing Stream and a filename.
   */
  public TeePrintStream(PrintStream os, String fn) throws IOException {
    this(os, fn, true);
  }

  /* Construct a TeePrintStream given an existing Stream, a filename,
   * and a boolean to control the flush operation.
   */
  public TeePrintStream(PrintStream orig, String fn, boolean flush)
  throws IOException {
    this(orig, new FileOutputStream(fn), flush);
    fileName = fn;
  }

  /** Return true if either stream has an error. */
  public boolean checkError() {
```

```
    return parent.checkError() || super.checkError();
  }

  /** override write(). This is the actual "tee" operation. */
  public void write(int x) {
    parent.write(x);  // "write once;
    super.write(x);    // write somewhere else."
  }

  /** override write(). This is the actual "tee" operation. */
  public void write(byte[] x, int o, int l) {
    parent.write(x, o, l);  // "write once;
    super.write(x, o, l);  // write somewhere else."
  }

  /** Close both streams. */
  public void close() {
    parent.close();
    super.close();
  }

  /** Flush both streams. */
  public void flush() {
    parent.flush();
    super.flush();
  }
}
```

It's worth mentioning that I do *not* need to override all the polymorphic forms of `print()` and `println()`. Because these all ultimately use one of the forms of `write()`, if you override the `print` and `println` methods to do the `tee`-ing as well, you can get several additional copies of the data written out.

10.12 Reading/Writing a Different Character Set

Problem

You need to read or write a text file using a particular encoding.

Solution

Convert the text to or from internal Unicode by specifying a converter when you construct an `InputStreamReader` or `PrintWriter`.

Discussion

Classes `InputStreamReader` and `OutputStreamWriter` are the bridge from byte-oriented `Streams` to character-based `Readers`. These classes read or write bytes and

translate them to or from characters according to a specified character encoding. The UTF-16 character set used inside Java (`char` and `String` types) is a 16-bit character set. But most character sets—such as ASCII, Swedish, Spanish, Greek, Turkish, and many others—use only a small subset of that. In fact, many European language character sets fit nicely into 8-bit characters. Even the larger character sets (script-based and pictographic languages) don't all use the same bit values for each particular character. The encoding, then, is a mapping between Java characters and an external storage format for characters drawn from a particular national or linguistic character set.

To simplify matters, the `InputStreamReader` and `OutputStreamWriter` constructors are the only places where you can specify the name of an encoding to be used in this translation. If you do not specify an encoding, the platform's (or user's) default encoding is used. `PrintWriters`, `BufferedReaders`, and the like all use whatever encoding the `InputStreamReader` or `OutputStreamWriter` class uses. Because these bridge classes only accept `Stream` arguments in their constructors, the implication is that if you want to specify a nondefault converter to read or write a file on disk, you must start by constructing not a `FileReader` or `FileWriter`, but a `FileInputStream` or `FileOutputStream`:

```
// io/UseConverters.java
BufferedReader fromKanji = new BufferedReader(
    new InputStreamReader(new FileInputStream("kanji.txt"), "EUC_JP"));
PrintWriter toSwedish = new PrinterWriter(
    new OutputStreamWriter(new FileOutputStream("sverige.txt"), "Cp278"));
```

Not that it would necessarily make sense to read a single file from Kanji and output it in a Swedish encoding. For one thing, most fonts would not have all the characters of both character sets; and, at any rate, the Swedish encoding certainly has far fewer characters in it than the Kanji encoding. A list of the supported encodings is also in the JDK documentation, in the file *docs/guide/internat/encoding.doc.html*. A more detailed description is found in Appendix B of *Java I/O*.

10.13 Those Pesky End-of-Line Characters

Problem

You really want to know about end-of-line characters, particulary for cross-platform file exchange.

Solution

Use `\r` and `\n` in whatever combination makes sense.

Discussion

If you are reading text (or bytes containing ASCII characters) in line mode using the `readLine()` method, you'll never see the end-of-line characters, and the same applies if you're using a `PrintWriter` with its `println()` method. Thus you won't be cursed with having to figure out whether `\n`, `\r`, or `\r\n` appears at the end of each line.

If you want that level of detail, you have to read the characters or bytes one at a time, using the `read()` methods. The only time I've found this necessary is in networking code, where some of the line-mode protocols assume that the line ending is `\r\n`. Even here, though, you can still work in line mode. When writing, pass `\r\n` into the `print()` (not dealing with the characters):

```
outputSocket.print("HELO " + myName + "\r\n");
String response = inputSocket.readLine();
```

For the curious, the strange spelling of "hello" is used in SMTP, the mail-sending protocol, where commands are four letters.

10.14 Beware Platform-Dependent File Code

Problem

Chastened by the previous recipe, you now wish to write only platform-independent code.

Solution

Use `readLine()` and `println()`. Avoid use of `\n` by itself.

Discussion

As mentioned in Recipe 10.13, if you just use `readLine()` and `println()`, you won't have to think about the line endings. But a particular problem, especially for former programmers of C and related languages, is using the `\n` character in text strings to mean a newline. What is particularly distressing about this code is that it works—sometimes—usually on the developer's own platform. But it will probably fail someday, on some other system:

```
String myName;
public static void main(String[] argv) {
  BadNewline jack = new BadNewline("Jack Adolphus Schmidt, III");
  System.out.println(jack);
}
/**
 * DON'T DO THIS. THIS IS BAD CODE.
 */
```

```
public String toString() {
  return super.toString() + "\n" + myName;
}

// The obvious Constructor is not shown for brevity; it's in the code
```

The real problem is not that it fails on some platforms, though. What's really wrong is that it mixes formatting and I/O, or tries to. Don't mix line-based display with `toString()`; avoid *multiline strings*—output from `toString()` or any other string-returning method. If you need to write multiple strings, then say what you mean:

```
String myName;
public static void main(String[] argv) {
  GoodNewline jack = new GoodNewline("Jack Adolphus Schmidt, III");
  jack.print(System.out);
}

protected void print(PrintStream out) {
  out.println(toString());  // classname and hashcode
  out.println(myName);    // print name  on next line
}
```

Alternatively, if you need multiple lines, you could return an array or `List` of strings.

10.15 Reading and Writing JAR or ZIP Archives

Problem

You need to create and/or extract from a JAR archive or a file in the well-known ZIP archive format, as established by PKZIP and used by Unix zip/unzip and WinZip.

Solution

You could use the `jar` program in the JDK because its file format is identical to the ZIP format, with the addition of a *META-INF* directory for additional metadata. But because this is a book about programming, you are probably more interested in the `ZipFile` and `ZipEntry` classes and the stream classes to which they provide access.

Discussion

The class `java.util.zip.ZipFile` allows you to read the contents of a JAR or ZIP-format file. When constructed, it creates a series of `ZipEntry` objects, one to represent each entry in the archive. In other words, the `ZipFile` represents the entire archive, and the `ZipEntry` represents one entry, or one file that has been stored (and compressed) in the archive. The `ZipEntry` has methods like `getName()`, which returns the name that the file had before it was put into the archive, and `getInputStream()`, which gives you an `InputStream` that will transparently uncompress the archive entry

by filtering it as you read it. To create a `ZipFile` object, you need either the name of the archive file or a `File` object representing it:

```
ZipFile zippy = new ZipFile(fileName);
```

To see whether a given file is present in the archive, you can call the `getEntry()` method with a filename. More commonly, you'll want to process all the entries; for this, use the `ZipFile` object to get a list of the entries in the archive, in the form of an `Enumeration` (see Recipe 7.7), as is done here:

```
Enumeration all = zippy.entries( );
while (all.hasMoreElements( )) {
    ZipEntry entry = (ZipEntry)all.nextElement( );
    ...
}
```

We can also get a `Stream<JarEntry>` using the `ZipFile.stream()` method. The following one-liner will list the contents of a `ZipFile`:

```
zippy.stream().forEach(System.out::println);
```

We can then process each entry as we wish. A simple listing program could be this:

```
if (entry.isDirectory( ))
    println("Directory: " + e.getName( ));
else
    println("File: " + e.getName( ));
```

A fancier version would extract the files. The program in Example 10-11 does both: it lists the files by default, but with the `-x` (extract) switch, it actually extracts the files from the archive.

Example 10-11. main/src/main/java/io/UnZip.java

```
public class UnZip {
  /** Constants for mode listing or mode extracting */
  public static enum Mode {
    LIST,
    EXTRACT;
  }
  /** Whether we are extracting or just printing TOC */
  protected Mode mode = Mode.LIST;

  /** The ZipFile that is used to read an archive */
  protected ZipFile zippy;

  /** The buffer for reading/writing the ZipFile data */
  protected byte[] b = new byte[8092];

  /** Simple main program, construct an UnZipper, process each
   * .ZIP file from argv[] through that object.
   */
```

```
public static void main(String[] argv) {
  UnZip u = new UnZip();

  for (int i=0; i<argv.length; i++) {
    if ("-x".equals(argv[i])) {
      u.setMode(Mode.EXTRACT);
      continue;
    }
    String candidate = argv[i];
    // System.err.println("Trying path " + candidate);
    if (candidate.endsWith(".zip") ||
      candidate.endsWith(".jar"))
        u.unZip(candidate);
    else System.err.println("Not a zip file? " + candidate);
  }
  System.err.println("All done!");
}

/** Set the Mode (list, extract). */
protected void setMode(Mode m) {
  mode = m;
}

/** Cache of paths we've mkdir()ed */
protected SortedSet<String> dirsMade;

/** For a given Zip file, process each entry. */
public void unZip(String fileName) {
  dirsMade = new TreeSet<String>();
  try {
    zippy = new ZipFile(fileName);
    @SuppressWarnings("unchecked")
    Enumeration<ZipEntry> all = (Enumeration<ZipEntry>) zippy.entries();
    while (all.hasMoreElements()) {
      getFile((ZipEntry)all.nextElement());
    }
  } catch (IOException err) {
    System.err.println("IO Error: " + err);
    return;
  }
}

protected boolean warnedMkDir = false;

/** Process one file from the zip, given its name.
 * Either print the name, or create the file on disk.
 */
protected void getFile(ZipEntry e) throws IOException {
  String zipName = e.getName();
  switch (mode) {
  case EXTRACT:
    if (zipName.startsWith("/")) {
```

```
        if (!warnedMkDir)
          System.out.println("Ignoring absolute paths");
        warnedMkDir = true;
        zipName = zipName.substring(1);
      }
      // if a directory, just return. We mkdir for every file,
      // since some widely used Zip creators don't put out
      // any directory entries, or put them in the wrong place.
      if (zipName.endsWith("/")) {
        return;
      }
      // Else must be a file; open the file for output
      // Get the directory part.
      int ix = zipName.lastIndexOf('/');
      if (ix > 0) {
        String dirName = zipName.substring(0, ix);
        if (!dirsMade.contains(dirName)) {
          File d = new File(dirName);
          // If it already exists as a dir, don't do anything.
          if (!(d.exists() && d.isDirectory())) {
            // Try to create the directory, warn if it fails.
            System.out.println("Creating Directory: " + dirName);
            if (!d.mkdirs()) {
              System.err.println(
              "Warning: unable to mkdir " + dirName);
            }
            dirsMade.add(dirName);
          }
        }
      }
      System.err.println("Creating " + zipName);
      FileOutputStream os = new FileOutputStream(zipName);
      InputStream  is = zippy.getInputStream(e);
      int n = 0;
      while ((n = is.read(b)) >0)
        os.write(b, 0, n);
      is.close();
      os.close();
      break;
    case LIST:
      // Not extracting, just list
      if (e.isDirectory()) {
        System.out.println("Directory " + zipName);
      } else {
        System.out.println("File " + zipName);
      }
      break;
    default:
      throw new IllegalStateException("mode value (" + mode + ") bad");
    }
  }
}
```

To *write* a `ZipFile`, use the `ZipOutputStream` class. This is demonstrated in Example 10-12. The program creates a new `ZipOutputStream` for the file *temp.zip*, creates two `ZipEntry` elements, and uses each of them to write some text.

Example 10-12. main/src/main/java/io/WriteZipFile.java

```
class WriteZipFile {

  public static final String FILENAME = "temp.zip";

  public static void main(String[] args) throws Exception {
    File file = new File(FILENAME);
    ZipOutputStream zf = new ZipOutputStream(new FileOutputStream(file));
    zf.putNextEntry(new ZipEntry("foo.bar.txt"));
    zf.write("Hello\n".getBytes());
    zf.putNextEntry(new ZipEntry("WriteZipFile.java"));
    Files.copy(
        Path.of("WriteZipFile.java"), zf);
    zf.closeEntry();
    zf.close();
    System.out.println("Written to " + FILENAME);
  }
}
```

Running the program creates the file, as verified by the system `unzip` command:

```
$ java src/main/java/io/WriteZipFile.java
$ unzip -t temp.zip
Archive:  temp.zip
    testing: foo.bar.txt              OK
    testing: WriteZipFile.java        OK
No errors detected in compressed data of temp.zip.
$ unzip temp.zip foo.bar.txt
Archive:  temp.zip
  inflating: foo.bar.txt
$ cat foo.bar.txt
Hello
$
```

See Also

People sometimes confuse the ZIP archive file format with the similarly named gzip compression format. Gzip-compressed files can be read or written with the `GZip InputStream` and `GZipOutputStream` classes from java.io.

10.16 Reading Files in a Filesystem-Neutral Way with getResource() and getResourceAsStream()

Problem

You want to load objects or files without referring to their absolute location in the filesystem. This is commonly used to package a file such as a configuration file into a JAR file and then access it from the code in the same file. It can also be used to provide a file (or different version of a file) in unit tests.

Solution

Use `getClass()` or `getClassLoader()` and either `getResource()` or `getResourceAsStream()`.

Discussion

There are three varieties of `getResource()` methods, some of which exist (with the exact same signature) both in the `Class` class (see Chapter 17) and in the `ClassLoader` class (see Recipe 17.9). The methods in `Class` delegate to the `ClassLoader`, so there is little difference between them. The methods are summarized in Table 10-8.

Table 10-8. The `getResource` methods

Method signature	In `Class`	In `ClassLoader`
`public InputStream getResourceAsStream(String);`	Y	Y
`public URL getResource(String);`	Y	Y
`public Enumeration<URL> getResources(String) throws IOException;`	N	Y

The first method is designed to quickly and easily locate a resource, or file, on your `CLASSPATH`. Using the `Class` version, or the other version with a standard `ClassLoader` implementation, the resource can be a physical file or a file inside a JAR file. If you define your own `ClassLoader`, your imagination is the limit, as long as it can be represented as an `InputStream`.

The common usage is shown here:

```
InputStream is = getClass().getResourceAsStream("foo.properties");
// check for null, then do something with the InputStream...
```

The second method returns a `URL`, which can be interpreted in various ways (see the discussion of reading from a URL in Recipe 14.1).

The third method, only usable with a `ClassLoader` instance, returns an `Enumeration` of `URL` objects. This is intended to return all the resources that match a given string; remember that a `CLASSPATH` can consist of an arbitrary number of directories and/or JAR files, so this method will search all of them. This is useful for finding a series of configuration files and merging them, perhaps. Or for finding out whether there is more than one resource/file of a given name on your `CLASSPATH`.

Note that the resource name can be given as either a relative path or as an absolute path. Assuming you are using Maven (see Recipe 2.4), then for the absolute path, place the file relative to the *src/main/resources/* directory. For the relative path, place the file in the same directory as your source code. The same rules apply in an IDE, assuming you have configured your IDE to treat *src/main/java* and *src/main/resources* as source folders. The idea is that resource files get copied to your `CLASSPATH` folder. For example, we have two resource files, *src/main/resources/getresourcedemo1.txt* and *src/main/java/io/myconfig.properties*, and the project is configured as described. We then execute the program shown in Example 10-13 in two different ways.

Example 10-13. main/src/main/java/io/GetResourceDemo.java

```
package io;

import java.io.*;

public class GetResourceDemo {
  private static final String MY_CONFIG_FILE = "myconfig.properties";

  public static void main(String[] args) throws Exception {
    Class<?> c = GetResourceDemo.class;

    // By file name
    System.out.println("Class is " + c.getName());
    final String pathName1 = "/getresourcedemo1.txt";
    System.out.println("pathName1 = " + pathName1);
    InputStream isOne = c.getResourceAsStream(pathName1);
    System.out.println("InputStream One = " + isOne);
    displayContents(isOne); // simple convenience routing

    // By package-relative name
    final String name = c.getPackage().getName();
    final String fullPathName = "/" + name.replace('.', '/') + "/" + MY_CONFIG_FILE;
    System.out.println("pathName2 = " + fullPathName);
    InputStream isTwo = c.getResourceAsStream(fullPathName);
    System.out.println("InputStream Two = " + isTwo);
    displayContents(isTwo);
  }
```

Build tools and IDEs will normally copy files from *src/main/resources* into their generated class folder, but they will not normally copy resource files found under

src/main/java. Thus two invocations of the program in Example 10-13 show two different results:

```
# Show locations of the two config files
$ find . -name getresourcedemo1.txt -o -name myconfig.properties
./src/main/java/io/myconfig.properties
./src/main/resources/getresourcedemo1.txt
./target/classes/getresourcedemo1.txt
$

# Run from within an IDE
Class is io.GetResourceDemo
pathName1 = /getresourcedemo1.txt
InputStream One = java.io.BufferedInputStream@621be5d1
This is a demo file for the GetResourceDemo program.
pathName2 = /io/myconfig.properties
InputStream Two = null

# Run from command line
$ cd src/main/java
/home/ian/git/javasrc/main/src/main/java
$ java io/GetResourceDemo.java
Class is io.GetResourceDemo
pathName1 = /getresourcedemo1.txt
InputStream One = null
pathName2 = /io/myconfig.properties
InputStream Two = java.io.BufferedInputStream@2ed0fbae
This is my config file, which is my own.
$
```

With either format, `getResource()` and `getResourceAsStream()` will return `null` if they don't find the resource; you should always check for `null` to guard against faulty deployment. If it doesn't find anything matching, `getResources()` will return an empty `Enumeration`.

If the file path has a package name, the name you pass into any of the `getResource` methods should have a slash in place of each period, reflecting how Java stores classes in subdirectories.

10.17 Creating a Transient/Temporary File

Problem

You need to create a file with a unique temporary filename and/or or arrange for a file to be deleted when your program is finished.

Solution

Use the `java.nio.file.Files` `createTempFile()` or `createTempDirectory()` method. Use one of several methods to ensure your file is deleted on exit.

Discussion

The `Files` class has static methods for creating temporary files and directories. Note that a temporary file in this context is not deleted automatically; it is simply created in a directory that is set aside for temporary files on that operating system (e.g., */tmp* on Unix). Here are the static `Files` methods for creating temporary files and directories:

`Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)`
: Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name

`Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`
: Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name

`Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)`
: Creates a new directory in the specified directory, using the given prefix to generate its name

`Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`
: Creates a new directory in the default temporary-file directory, using the given prefix to generate its name

The file attributes are discussed in "Understanding I/O Options: StandardOpenOptions, FileAttribute, PosixFileAttribute, and More" on page 370.

There are various ways to arrange for a file to be deleted automatically. One option is to use the legacy `java.io.File` class, which has an explicit `deleteOnExit()` method. This arranges for any file (no matter how it was created) to be deleted if it still exists when the program exits. Here we arrange for a backup copy of a program to be deleted on exit, and we also create a temporary file and arrange for it to be removed on exit. Both files are gone after the program runs:

```
public class TempFiles {
  public static void main(String[] argv) throws IOException {

    // 1. Making an existing file temporary
    // Construct a File object for the backup created by editing
    // this source file. The file probably already exists.
    // My editor creates backups by putting ~ at the end of the name.
```

```
        File bkup = new File("Rename.java~");
        // Arrange to have it deleted when the program ends.
        bkup.deleteOnExit();

        // 2. Create a new temporary file.

        // Make a file foo.tmp, in the default temp directory.
        Path tmp = Files.createTempFile("foo", "tmp");
        // Report on the filename that it made up for us.
        System.out.println("Your temp file is " + tmp.normalize());
        // Arrange for it to be deleted at exit.
        tmp.toFile().deleteOnExit();
        // Now do something with the temporary file, without having to
        // worry about deleting it later.
        writeDataInTemp(tmp);
    }

    public static void writeDataInTemp(Path tempFile) throws IOException {
        // This version is dummy. Use your imagination.
        Files.writeString(tempFile, "This is a temp file");
    }
}
```

When run on a Unix system, the program looks like this, proving that the file was created but removed when the JVM exited:

```
$ java TempFiles.java
Your temp file is /tmp/foo842332191021505468 9tmp
$ ls -l /tmp/foo842332191021505468 9tmp
ls: /tmp/foo842332191021505468 9tmp: No such file or directory
$
```

The `createTempFile()` method is like `createNewFile()` (see Recipe 10.3) in that it does create the file. You should also be aware that, should the JVM terminate abnormally, the deletion probably will not occur. There is no way to undo the setting of `deleteOnExit()` short of renaming the file or something drastic like powering off the computer before the program exits.

Another way to arrange for any file to be deleted when you are finished with it is to create it with the `DELETE_ON_CLOSE` option (see Table 10-5) so it will be deleted when you close the file.

A third, less likely, method is to instead use a JVM shutdown hook. `DELETE_ON_CLOSE` is probably the best option, particularly in a long-running application, like most server-side apps. In these situations, the server could be running for weeks, months, or even years. In the meantime all the temp files would accumulate and the JVM would accumulate a large list of deferred work that it needs to perform upon shutdown. You'd probably run out of disk space or server memory or some other resource. For most long-running apps of this kind, it's better to use `DELETE_ON_CLOSE`

or even the explicit `delete()` operation. Another alternative is to use a scheduler service to periodically trigger removal of old temporary files.

10.18 Getting the Directory Roots

Problem

You want information about the top-level directories, such as *C:* and *D:* on Windows.

Solution

Use the static method `FileSystems.getDefault().getRootDirectories()`, which returns an `Iterable` of `Path` objects, one for each root directory. You can print them or do other operations on them.

Discussion

Operating systems differ in how they organize filesystems out of multiple disk drives or partitions. Microsoft Windows has a low-level device-oriented approach in which each disk drive has a root directory named *A:* for the first floppy drive (if you still have one!), *C:* for the first hard drive, and other letters for DVD-ROM (or CD-ROM) and network drives. This approach requires you to know the physical device on which a file is stored. Unix, Linux, and macOS have a higher-level approach with a single root directory /, and different disks or partitions are *mounted*, or connected, into a single unified tree. This approach sometimes requires you to figure out where a device file is mounted. Perhaps neither is easier, though the Unix approach is a bit more consistent. Either way, Java makes it easy for you to get a list of the roots.

The static method `FileSystems.getDefault().getRootDirectories()` returns an `Iterable<Path>` containing the available filesystem roots for whatever platform you are running on. Here is a short program to list these:

```
FileSystems.getDefault().getRootDirectories().forEach(System.out::println);
```

```
C:> java dir_file.DirRoots
A:\
C:\
D:\
C:>
```

Run on Microsoft Windows, the program listed my floppy drive (even though the floppy drive was not only empty, but also left at home while I wrote this recipe on my notebook computer in my car in a parking lot), the hard disk drive, and the DVD-ROM drive.

On Unix there is only one root directory:

```
$ java dir_file.DirRoots
/
$
```

One thing that is left out of the list of roots is the so-called *UNC filename*. UNC filenames are used on some Microsoft platforms to refer to a network-available resource that hasn't been mounted locally on a particular drive letter. If your system still uses these, be aware they will not show up in the `listDirectoryRoots()` output.

10.19 Using the File Watcher Service to Get Notified About File Changes

Problem

You want to be notified when some other application updates one or more of the files in which you are interested.

Solution

Use the `java.nio.file.WatchService` to get notified of changes to files automatically, instead of having to examine the files periodically.

Discussion

It is fairly common for a large application to want to be notified of changes to files, without having to go and look at them periodically. For example, an IDE wants to know when files were modified by an external editor or a build script. A Java Enterprise server wants to know if the class file for components got updated. Many modern operating systems have had this capability for some time, and now it is available in Java.

These are the basic steps to using the `WatchService`:

1. Create a `Path` object representing the directory you want to watch.
2. Get a `WatchService` by calling, for example, `FileSystems.getDefault().new WatchService()`.
3. Create an array of `Kind` enumerations for the things you want to watch (in our example we watch for files being created or modified).
4. Register the `WatchService` and the `Kind` array onto the `Path` object.

5. From then on, you wait for the watcher to notify you. A typical implementation is to enter a `while (true)` loop calling the `WatchService`'s `take()` method to get an event and interpret the events to figure out what just happened.

Example 10-14 is a program that does just that. In addition, it starts another thread to actually do some filesystem operations so that you can see the `WatchService` operating.

Example 10-14. main/src/main/java/nio/FileWatchServiceDemo.java

```
public class FileWatchServiceDemo {

  final static String TEMP_DIR_PATH = "/tmp";
  static final String FILE_SEMA_FOR = "MyFileSema.for";
  final static Path SEMAPHORE_PATH = Path.of(TEMP_DIR_PATH ,FILE_SEMA_FOR);
  static volatile boolean done = false;
  final static ExecutorService threadPool = Executors.newSingleThreadExecutor();

  public static void main(String[] args) throws Throwable {
    String tempDirPath = "/tmp";
    System.out.println("Starting watcher for " + tempDirPath);
    System.out.println("Semaphore file is " + SEMAPHORE_PATH);
    Path p = Paths.get(tempDirPath);
    WatchService watcher =
      FileSystems.getDefault().newWatchService();
    Kind<?>[] watchKinds = { ENTRY_CREATE, ENTRY_MODIFY };
    p.register(watcher, watchKinds);
    threadPool.submit(new DemoService());
    while (!done) {
      WatchKey key = watcher.take();
      for (WatchEvent<?> e : key.pollEvents()) {
        System.out.println(
          "Saw event " + e.kind() + " on " +
          e.context());
        if (e.context().toString().equals(FILE_SEMA_FOR)) {
          System.out.println("Semaphore found, shutting down watcher");
          done = true;
        }
      }
      if (!key.reset()) {
        System.err.println("WatchKey failed to reset!");
      }
    }
  }

  /**
   * Nested class whose only job is to wait a while, create a file in
   * the monitored directory, and then go away.
   */
  private final static class DemoService implements Runnable {
```

```
    public void run() {
      try {
        Thread.sleep(1000);
        System.out.println("DemoService: Creating file");
        Files.deleteIfExists(SEMAPHORE_PATH); // clean up from previous run
        Files.createFile(SEMAPHORE_PATH);
        Thread.sleep(1000);
        System.out.println("DemoService: Shutting down");
      } catch (Exception e) {
        System.out.println("Caught UNEXPECTED " + e);
      }
    }
  }
}
```

10.20 Walking a File Tree (like Find)

Problem

You need to work through some or all of the files and directories in a filesystem tree, that is, recursively visit all filesystem entries under a given starting point.

Solution

Use the `walk()` or `walkFileTree()` methods in the `Files` class.

Discussion

The `Files` class has four methods for walking a file tree. Two return a lazily populated `Stream<Path>`, and the other two invoke a callback `FileVisitor` for each file or directory found. As the name hints, this is an example of the GoF Visitor pattern. My `find` implementation uses the first method; the four are summarized in Table 10-9.

Table 10-9. `Files` tree walk methods

Return	Signature
`Stream<Path>`	`walk(Path start, FileVisitOption... options)`
`Stream<Path>`	`walk(Path start, int maxDepth, FileVisitOption... options)`
`Path`	`walkFileTree(Path start, FileVisitor<? super Path> visitor)`
`Path`	`walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth, FileVisitor<? super Path> visitor)`

Using the `walk()` methods is as simple as this:

```
    Files.walk(startingPath).forEach(path -> {
            // Do something with Path path; might be a file, directory, or other...
    }
```

That code is near the start of the `startWalkingAt()` method in my *Find.java* file (in the *javasrc* repo, in the file *main/src/main/java/dir_file/Find.java*).

The `walkFileTree()` methods both require a `FileVisitor()`, a simple interface with four callback methods, summarized in Table 10-10.

Table 10-10. `FilesVisitor` interface

Method signature	When invoked
`FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`	Called before each directory is started
`FileVisitResult postVisitDirectory(T dir, IOException exc)`	Called when all the entries in a directory (including subdirectories, recursively) have been processed
`FileVisitResult visitFile(T file, BasicFileAttributes attrs)`	Called for each file in a directory
`FileVisitResult visitFileFailed(T file, IOException exc)`	Called for a file that couldn't be visited; the `IOException` will explain the failure.

Callers of these methods can implement the `FileVisitor` interface, or if only some methods are needed, subclass the `SimpleFileVisitor` class. These methods are demonstrated in Example 10-15.

Example 10-15. main/src/main/java/dir_file/WalkTreeDemo.java

```
public class WalkTreeDemo {

  public static void main(String[] args) throws IOException {
    String dir = args.length == 0 ? "." : args[0];
    Files.walkFileTree(Path.of(dir), myVisitor);
  }

  static final FileVisitor myVisitor = new FileVisitor() {
    @Override
    public FileVisitResult preVisitDirectory(Object dir,
      BasicFileAttributes attrs) throws IOException {
      System.out.println("Starting Directory " + dir);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Object dir,
      IOException exc) throws IOException {
      System.out.println("Finished Directory " + dir +
        " " + (exc != null? exc : ""));
      return FileVisitResult.CONTINUE;
    }
```

```
    @Override
    public FileVisitResult visitFile(Object file,
      BasicFileAttributes attrs) throws IOException {
      System.out.println("Visiting File " + file);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Object file,
      IOException exc) throws IOException {
      System.out.println(
        MessageFormat.format("FAILURE visiting {0} ({1})", file, exc));
      return FileVisitResult.CONTINUE;
    }
  };
}
```

My simplified Find program implements a subset of the Windows Find Files dialog or the Unix `find` command. It has most of the structure needed to build a more complete version of either of these. The program accepts the following options from the standard Unix `find` (with limits):

`-n` *name*
: Name to look for. Can include shell wildcards if quoted from the shell.

`-s` *size*
: Size of file to look for. Can prefix with a plus sign to indicate greater than or a minus sign to indicate less than.

`-a`, `-o`
: *And* or *or*, but currently only one of these, between an `-n` and an `-s`.

The entire Find program is too long to include in the book, but can be found in the *javasrc* repo, in the files *main/src/main/java/dir_file* directory. The files are also available at *https://github.com/IanDarwin/javasrc/blob/main/main/src/main/java/dir_file*.

CHAPTER 11

Threaded Java

11.0 Introduction

We live in a world of multiple activities. A person may be talking on the phone while doodling or reading an email. A multifunction office machine may scan one fax while receiving another and printing a document from somebody's computer. We expect the GUI programs we use to be able to respond to a menu while updating the screen. But ordinary computer programs can do only one thing at a time. The conventional computer programming model—that of writing one statement after another, punctuated by repetitive loops and binary decision making—is sequential at heart.

Sequential processing is straightforward but not as efficient as it could be. To enhance performance, Java offers *threading*, the capability to handle multiple flows of control within a single application or process. Java provides thread support and, in fact, requires threads: the Java runtime itself is inherently multithreaded. For example, window system action handling and Java's garbage collection—that miracle that lets us avoid having to free everything we allocate, as others must do when working in languages at or below C level—run in separate threads.

Just as multitasking allows a single operating system to give the appearance of running more than one program at the same time on a single-processor computer, multithreading can allow a single program or process to give the appearance of working on more than one thing at the same time. Multithreading leads to more interactive graphics and more responsive GUI applications (the program can draw in a window while responding to a menu, with both activities occurring more or less independently), more reliable network servers (if one client does something wrong, the server continues communicating with the others), and so on.

Note that I did not say "multiprocessing;" the term multi-tasking is sometimes erroneously called multiprocessing, but the latter term in fact refers to a different issue:

it's the case of two or more CPUs running under a single operating system. Multiprocessing per se is nothing new: IBM mainframes did it in the 1970s, Sun SPARCstations did it in the 1980s, and Intel PCs did it starting in the 1990s. Since the mid-2010s, it has become increasingly hard to buy a single-processor computer packaged inside anything larger than a wristwatch. True multiprocessing allows you to have more than one process running concurrently on more than one CPU. Java's support for threading includes multiprocessing, as long as the operating system supports it. Consult your system documentation for details.

Though most modern operating systems provide threads, Java was the first mainstream programming language to have intrinsic support for threaded operations built right into the language. The semantics of `java.lang.Object`, of which all objects are instances, includes the notion of monitor locking of objects, and some methods (`notify`, `notifyAll`, `wait`) are meaningful only in the context of a multithreaded application. Java also has language keywords such as `synchronized` to control the behavior of threaded applications.

In modern Java there are two types of threads, *platform threads* and *virtual threads*. As the name implies, the former are coupled one-to-one with platform (operating system) threads, which the latter are multiplexed among a small pool of platform threads.

Threads can have a name associated with them, and a priority. Platform threads can also be *daemon* or *non-daemon*. When a normal Java application is started, there is one non-daemon thread (e.g., the thread that called the application's `main()` method from the `java` command). The application will shut down when all running non-daemon threads have terminated. Non-daemon threads are used, for example, to handle mouse and keyboard events in a GUI application, such that the `main()` method can end or `return` without causing the application to terminate. Platform threads have a thread priority and are members of a thread group. Virtual threads are always daemon threads. Recipe 11.2 provides coverage specific to virtual threads.

Now that the world has had years of experience with threaded Java, experts have started building better ways of writing threaded applications. The concurrency utilities, originally specified in JSR-166 and included in all modern Java releases, are heavily based on the util.concurrent package by Professor Doug Lea of the Computer Science Department at the SUNY Oswego. This package aims to do for the difficulties of threading what the Collections Framework (see Chapter 7) did for structuring data. This is no small undertaking, but they pulled it off.

The java.util.concurrent package includes several main sections:

- `Executors`, thread pools (`ExecutorServices`), and `Futures`/`CompletableFutures`
- `Queues` and `BlockingQueues`

- Locks and conditions, with JVM support for faster locking and unlocking
- Synchronizers, including `Semaphores` and `Barriers`
- Atomic variables

In this chapter, I will focus on the first set of these, thread pools and `Futures`.

An implementation of the `Executor` interface is, as the name implies, a class that can execute code for you. The code to be executed can be the familiar `Runnable` or a new interface `Callable`. One common type of `Executor` is a *thread pool*. The `Future` interface represents the future state of something that has been started; it has methods to wait until the result is ready. A `CompletableFuture` is an implementation of `Future` that adds many additional methods for chaining `CompletableFutures` and post-applied methods. These brief definitions are simplifications. Addressing all the issues is beyond the scope of this chapter, but the chapter provides examples of the most important parts.

11.1 Running Code in a Different Thread

Problem

You need to write a threaded application. Threads allow your program to do more than one thing at a time, such as servicing multiple users, multiple connections, or in a desktop application, responding to UI events while loading a file, or in a video game, moving multiple objects around the screen.

Solution

Write code that implements `Runnable`; pass it to an `Executor`, or instantiate a `Thread` and start it.

Discussion

There are several ways to implement threading, and they all require you to implement the `Runnable` or `Callable` interface. `Runnable` has only one method, and it returns no value; this is its signature:

```
public interface java.lang.Runnable {
  public abstract void run();
}
```

Similarly, `Callable` has only one method, but the `call()` method returns a specific type so the interface has a type parameter (`V` here, for "value"):

```
public interface java.util.concurrent.Callable<V> {
  public abstract V call() throws Exception;
}
```

You must provide an implementation of the `run()` or `call()` method. There is nothing special to this method; it's an ordinary method and you could call it yourself. But if you did, what then? There wouldn't be the special magic that launches it as an independent flow of control, so it wouldn't run concurrently with your main program or flow of control. For this, you need to invoke the magic of thread creation.

The original approach to using threads, no longer generally recommended, is to create `Thread` objects directly and call their `start()` method, which would cause the thread to call the `run()` method after the new thread had been initialized. There was no support for the `Callable` interface in the original threads model. You create threads by doing one of the following:

- Subclass `java.lang.Thread` (which implements `Runnable`) and override the `run()` method.
- Create your `Runnable` and pass it into the `Thread` constructor.
- With Java 8+, you can use a lambda expression to implement `Runnable`, as shown in the introdcution to Chapter 9.

This approach is not recommended because of issues such as performance (`Thread` objects are expensive to create and tear down, and a thread is unusable once its `run()` method returns). This book no longer shows examples of doing so. There are some examples in the online source, in the *threads* directory; see especially *ThreadsDemo4*.

When you need to stop a thread, don't use the `Thread.stop()` method; instead, use a `boolean` tested at the top of the main loop in the `run()` method. That's because the `stop()` method is so drastic that it can never be made to behave reliably in a program with multiple active threads. That is why, when you try to use it, the compiler will generate deprecation warnings. The recommended method (for threads with a loop) is to use a `boolean` variable in the main loop of the `run()` method. Example 11-1 prints a message endlessly until its `shutDown()` method is called; it then sets the controlling variable `done` to `false`, which terminates the loop. This causes the `run()` method to return, ending its processing.

Example 11-1. main/src/main/java/threads/StopBoolean.java

```
public class StopBoolean {

  // Must be volatile to ensure changes visible to other threads.
  protected volatile boolean done = false;

  Runnable r = () -> {
```

```
    while (!done) {
      System.out.println("StopBoolean running");
      try {
        Thread.sleep(720);
      } catch (InterruptedException ex) {
        // nothing to do
      }
    }
    System.out.println("StopBoolean finished.");
  };

  public void shutDown() {
    System.out.println("Shutting down...");
    done = true;
  }

  public void doDemo() throws InterruptedException {
    ExecutorService pool = Executors.newSingleThreadExecutor();
    pool.submit(r);
    Thread.sleep(1000*5);
    shutDown();
    pool.shutdown();
    pool.awaitTermination(2, TimeUnit.SECONDS);
  }

  public static void main(String[] args) throws InterruptedException {
    new StopBoolean().doDemo();
  }
}
```

Instead of starting and stopping threads explicitly, the recommended way to perform threaded operations is to use a thread pool, the implementation of which is the java.util.concurrent package's `ExecutorService`. An `ExecutorService` is, as its name implies, a service class that can execute code for you. The code to be executed can be in a `Runnable` or a `Callable`. You obtain an `ExecutorService` by invoking a factory method on the `Executors` class. The code in Example 11-2 shows a simple example of a thread pool.

Example 11-2. main/src/main/java/threads/ThreadPoolDemo.java

```
    final ExecutorService pool = Executors.newFixedThreadPool(HOWMANY);
    List<Future<Integer>> futures = new ArrayList<>(HOWMANY);
    for (int i = 0; i < HOWMANY; i++) {
      Future<Integer> f = pool.submit(new DemoRunnable(i));
      System.out.println("Got 'Future' of type " + f.getClass());
      futures.add(f);
    }
    Thread.sleep(3 * 1000);
    done = true;
    for (Future<Integer> f : futures) {
```

```
      System.out.println("Result " + f.get());
    }
    pool.shutdown();
  }

  static class DemoRunnable implements Callable<Integer> {
    int time, numRuns;
    DemoRunnable(int t) { time = t; }

    @Override public Integer call() {
      while (!done) {
        System.out.println("Running " + Thread.currentThread());
        ++numRuns;
      }
      System.out.println("Stopping " + this);
      return numRuns;
    }
  }
```

This will print a series of lines like the following, showing the threads running interspersed:

```
Running Thread[pool-1-thread-3,5,main]
Running Thread[pool-1-thread-3,5,main]
Running Thread[pool-1-thread-1,5,main]
Running Thread[pool-1-thread-1,5,main]
...
Stopping threads.ThreadPoolDemo$DemoRunnable@2bdd2b00
Stopping threads.ThreadPoolDemo$DemoRunnable@3e413b00
Result 120056
Result 123719
...
```

Note that there are several submission methods, the first in the parent interface `Executor` and two more in `ExecutorService`:

```
public void execute(Runnable);
public Future<T> submit(Callable<T>);
public Future<T> submit(Runnable);
```

That is, `execute()` takes a `Runnable` and returns nothing, while the `submit()` methods both return a `Future<T>` (for the method `submit(Runnable)`, the type parameter `T` is always `java.lang.Void`).

When you are finished with the thread pool, you should call its `shutDown()` method.

Understanding Future and CompletableFuture

`Future` is a software analogue of the claim ticket you are given when you take laundry in to a dry cleaning service or take some item in to be repaired. It's an interface representing a claim on some deliverable that either will or will not be ready at some point in time, and that will hopefully change from not ready to ready "sometime." The following shows the important methods of the `Future` interface:

```
public interface java.util.concurrent.Future<V> {
    public abstract boolean isDone();
    public abstract V get() throws InterruptedException,
        java.util.concurrent.ExecutionException;
    public abstract V get(long timeout, java.util.concurrent.TimeUnit)
        throws InterruptedException,
        java.util.concurrent.ExecutionException,
        java.util.concurrent.TimeoutException;
    public abstract boolean cancel(boolean);
    public abstract boolean isCancelled();
}
```

The purpose of each method in this interface is shown here:

`isDone()`
: Returns true if the operation that will deliver the result has completed.

`get()`
: Will return the deliverable immediately if `isDone()` is true; else will block indefinitely until it becomes true.

`get(long, TimeUnit)`
: Will return the deliverable immediately if `isDone()` is true, blocking until it becomes true, or until the specified time has elapsed, in which case it will throw a `TimeoutException`. `TimeUnit` is an enum in `java.util.concurrent` with values for days, hours, minutes, seconds, etc.

`cancel(boolean)`
: Will cancel the operation if it hasn't started (if the `boolean` is false) or even if it is in process (if the `boolean` is true).

`isCancelled()`
: Returns true if the operation has been canceled.

`Future` is commonly returned from a thread pool `execute()` operation, as shown in Example 11-3.

Example 11-3. main/src/main/java/threads/FutureFromThreadpool.java

```
double d = 2;
Callable<Double> computeTotal = () -> d + d;
Future<Double> future = threadPool.submit(computeTotal);
```

```
while (!future.isDone()) {
  Thread.sleep(100);
}
double value = future.get();
process(value);
threadPool.shutdown();
```

There are many classes implementing `Future` in various parts of Java SE and Jakarta. The most general and powerful is `CompletableFuture<V>`, so called because you can control it by calling a `complete(V)` method at any time. This has far too many public methods (120) for a complete treatment here. In fairness, the number of methods is high because many methods can accept either a `Runnable` or a `Callable`, and many have multiple overloads (a plain one, one with `Async` appended to the name, and one with `Async` that lets you provide the `Executor`). I'll show examples of a few of these shortly.

You can create an empty `CompletableFuture` by calling a no-argument constructor, making this available to some calling code, and calling its `complete()` method when you have a result:

```
CompletableFuture<Integer> cf = new CompletableFuture<>();
// Do some work
cf.complete();
```

Many of the more interesting methods in `CompletableFuture` have to do with chaining operations. First, you can specify a function to be invoked automatically when the result is ready:

```
public CompletableFuture<Void> thenRun(java.lang.Runnable);
public CompletableFuture<Void> thenRunAsync(java.lang.Runnable);
public CompletableFuture<Void> thenRunAsync(java.lang.Runnable, Executor);
public <U> CompletableFuture<U> thenApply(
    Function<? super T,
        ? extends U>);
public <U> CompletableFuture<U> thenApplyAsync(
    Function<? super T,
        ? extends U>);
public <U> CompletableFuture<U> thenApplyAsync(
    Function<? super T,
        ? extends U>, Executor);
public CompletableFuture<Void> thenAccept(Consumer<? super T>);
public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T>);
public CompletableFuture<Void> thenAcceptAsync(
    Consumer<? super T>, Executor);
```

These methods will invoke the given `Runnable`, `Consumer`, or `Function` after the `Future` is completed. Each of these exists in the three forms as mentioned previously. The first method will run it on the same thread as the main task. The second method will run it in a default executor. The third allows you to provide your own `Executor`. Example 11-4 is a simple demo of creating a `CompletableFuture` and giving it a

`thenApply` method call and a final `run` method, both of which don't fire until the `Future` is completed.

Example 11-4. main/src/main/java/threads/CompletableFutureSimple.java

```
class CompletableFutureSimple {
  static String twice(String x) { return x + ' ' + x; }

  public static void main(String[] args) {
    CompletableFuture<String> cf = new CompletableFuture<>();
    cf.thenApply(x -> twice(x))
      .thenAccept(x -> System.out.println(x));
    // Possibly some computation going on here... Then:
    cf.complete("Hello");
  }
}
```

The online source includes *CompletableFutureDemo.java*, which offers some more sophisticated examples.

11.2 Using Virtual Threads for Better Performance

Problem

You want to experience faster performance of a multithreaded, I/O-bound process.

Solution

Use virtual threads.

Discussion

The original threading mechanism, standard since the early days of Java, had one major drawback: overhead. Each Java thread was tied to a single OS-level thread. When that thread was blocked, for example when doing input/output, the Java thread could not progress. Actually, some of the very early JVMs used a different model, called *green threads*, in which the Java threads were multiplexed onto system-level threads, initially because some of the target operating systems then in use didn't support application use of threads! This was abandoned after all the operating systems caught up. With virtual threads we see a return to this policy of multiplexing, but with two decades of experience in how to design it better. Now, the JVM will attach the virtual thread to an available OS-level thread. This has the consequence that a task running in a virtual thread may be attached to several different physical threads over its lifetime.

The nice thing about virtual threads is that, from a developer's point of view, they can most easily be used with what you already know. We mentioned earlier that thread pools are the recommended means of writing threaded code. That's still true with virtual threads. You just have to say:

```
ExecutorService pool = Executors.newVirtualThreadPerTaskExecutor();
```

And presto! Tasks you submit to `pool` will run on virtual threads.

Alternatively, if for some reason you don't want to use a thread pool, you can get virtual threads one at a time just by asking and providing a `Runnable`:

```
// Mainly for apps that only ever need one thread
Thread t = Thread.startVirtualThread(runnable);
```

Since the return type is `Thread` and there is no `Future` associated with a `Thread`, if you want to use a `Callable`, you must use the thread pool approach.

Both approaches are demonstrated in Example 11-5.

Example 11-5. main/src/main/java/threads/VirtualThreadsDemo.java

```
public static void main(String[] args) throws Exception {

  Runnable r = () -> {
    System.out.println("Hello from " + Thread.currentThread());
  };

  // Start vthreads with an ExecutorService
  try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, MAX).forEach(i -> executor.submit(r));
  }

  // Start vthreads directly
  IntStream.range(0,MAX).forEach(i -> Thread.startVirtualThread(r));

  Thread.sleep(100);
}
```

Note that `MAX` in the preceding example can be set to be quite large—you can create many thousands of virtual threads, whereas that many regular threads would probably bring the operating system to its (virtual) knees. An app with that many tasks happening at once would probably benefit from virtual threads.

Incidentally, when you want a `Callable` but have a `Runnable`, there are several overloads of a converter method in the `Executors` class, including:

```
static callable(Runnable task);
static <T> Callable<T> callable(Runnable task, T result);
```

These create a `Callable` wrapper that returns `null` in the first case or a value of the provided type in the second case.

Not all types of programs will benefit from switching to virtual threads. Programs that have a lot of threads waiting for I/O operations to complete are the most likely to see improvements. Determining whether virtual threads are faster requires careful measurement with a significant amount of input.

We didn't take the time to compare the implementations, as you'd need a larger sample to see the difference. You might look at *main/src/main/java/lang/Timer.java* for a skeleton program for timing comparisons.

11.3 Rendezvous and Timeouts

Problem

You need to know whether something finished or whether it finished in a certain length of time.

Solution

Start it in its own thread and call its `join()` method with or without a timeout value.

Discussion

The `join()` method of the target thread is used to suspend the current thread until the target thread is finished (that is, returns from its `run()` method). This method is overloaded; a version with no arguments waits forever for the thread to terminate, whereas a version with arguments waits up to the specified time. For a basic example, I create (and start!) a simple thread that just reads from the console terminal, and the main thread simply waits for it. When I run the program, it looks like this:

```
ian $ java threads.Join
Starting
Joining
Reading
hello from standard input # blocking wait for me to type this line
Thread Finished.
Main Finished.
ian $
```

Example 11-6 lists the code for the `join()` demo.

Example 11-6. main/src/main/java/threads/Join.java

```
public class Join {
  public static void main(String[] args) {
    System.out.println("Starting");
    Thread t = Thread.startVirtualThread(() -> {
        System.out.println("Reading");
        try {
          System.in.read();
        } catch (java.io.IOException ex) {
          System.err.println(ex);
        }
        System.out.println("Thread Finished.");
    });
    System.out.println("Joining");
    try {
      t.join();
    } catch (InterruptedException ex) {
      // should not happen:
      System.out.println("Who dares interrupt my sleep?");
    }
    System.out.println("Main Finished.");
  }
}
```

This code uses a `Runnable` lambda (see Recipe 11.1) in `Thread t`.

11.4 Synchronizing Threads with the synchronized Keyword

Problem

You need to protect certain data from access by multiple threads.

Solution

Use the `synchronized` keyword on the method or code you wish to protect.

Discussion

This keyword specifies that only one thread at a time is allowed to run the given method (or any other synchronized method in the same class) in a given object instance (for static methods, only one thread is allowed to run the method at a time). You can synchronize methods or smaller blocks of code. It is easier and safer to synchronize entire methods, but this can be more costly in terms of blocking threads that could run. To do so, you can simply add the `synchronized` keyword on the method. For example, many of the methods of `Vector` (see Recipe 7.5) are synchronized in

order to ensure that the vector does not become corrupted or give incorrect results when two threads update or retrieve from it at the same time.

Bear in mind that threads can be interrupted at almost any time, in which case control is given to another thread. Consider the case of two threads appending to a data structure at the same time. Let's suppose we have the same methods as `Vector`, but we're operating on a simple array. The `add()` method simply uses the current number of objects as an array index, then increments it:

```
public void add(Object obj) {
   data[max] = obj; ❶
   max = max + 1;   ❷
}
```

Threads A and B both want to call this method. Suppose that Thread A gets interrupted after ❶ but before ❷, and then Thread B gets to run.

❶ Thread B does ❶, overwriting the contents of `data[max]`; we've now lost all reference to the object that Thread A passed in!

❷ Thread B then increments `max` at ❷ and returns. Later, Thread A gets to run again; it resumes at ❷ and increments `max` past the last valid object. So not only have we lost an object, but we have an uninitialized reference in the array. This state of affairs is shown in Figure 11-1.

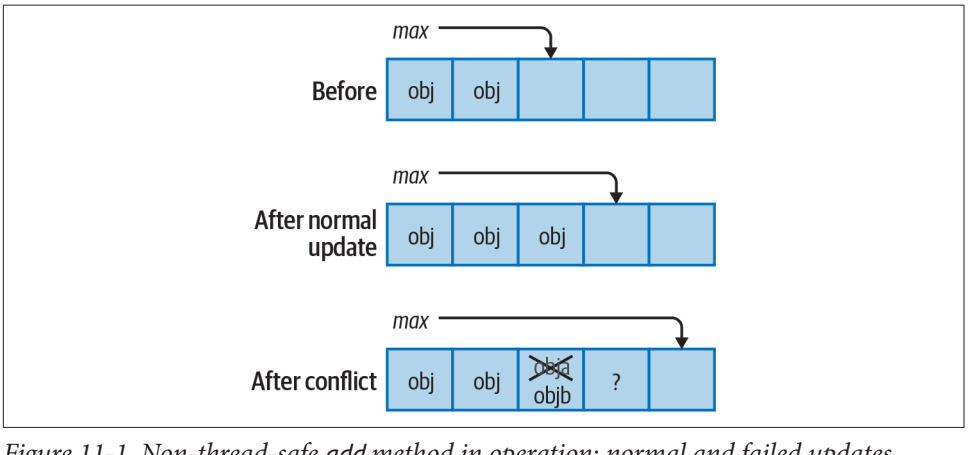


Figure 11-1. Non-thread-safe add method in operation: normal and failed updates

Now, you might think, "No problem, I'll just combine the two lines of code!":

```
data[max++] = obj;
```

As the game show host sometimes says, "Bzzzzt! Thanks for playing!" This change makes the code a bit shorter but has absolutely no effect on reliability. Interrupts

don't happen conveniently on Java statement boundaries; they can happen between any of the many JVM instructions that correspond to your program. The code can still be interrupted after the store and before the increment. The only good solution is to use proper synchronization.

Making the method `synchronized` means that any invocations of it will wait if one thread has already started running the method:

```
public synchronized void add(Object obj) {
    ...
}
```

Any time you wish to synchronize some code, but not an entire method, use the `synchronized` keyword on an unnamed code block within a method, like this:

```
public void add(Object obj) {
    synchronized (someObject) {
        // this code will execute in one thread at a time
    }
}
```

The choice of which object to synchronize on is up to you. Sometimes it makes sense to synchronize on the object containing the code (e.g., `synchronized(this) {…}`) or using a `synchronized` method. For synchronizing access to an `ArrayList`, it would make sense to use the `ArrayList` instance, like this:

```
synchronized(myArrayList) {
     if (myArrayList.indexOf(someObject) != -1) {
         // do something with it.
     } else {
         create an object and add it...
    }
}
```

The code in Example 11-7 shows an array being accessed in a loop by two threads. Two versions of the accessing code are shown, `runGood` and `runBad`. Only the former is synchronized, and only the former is reliable. The code synchronizes on the array, so any code accessing the array while either thread is still running should also synchronize on the array.

Example 11-7. main/src/main/java/threads/ArrayAdding.java

```
public class ArrayAdding {
  private static final int HOWMANY = 1000;
    private static int[] array;
  private static ExecutorService pool = Executors.newFixedThreadPool(2);

    static Runnable runBad = () -> {
        for (int i = 0; i < array.length; i++) {
            array[i] = array[i] + i;
```

```
        }
    };

  static Runnable runGood = () -> {
    synchronized(array) {
      for (int i = 0; i < array.length; i++) {
        array[i] = array[i] + i;
      }
    }
  };

    public static void main(String[] args) throws Exception {
    process("runGood", runGood);
    process("runBad", runBad);
  }

  static void process(String name, Runnable run) throws Exception {
    System.out.println("Starting: " + name);
    array = new int[HOWMANY];
        var t1 = pool.submit(runBad);
        var t2 = pool.submit(runBad);
    t1.get();
    t2.get();
    for (int i = 0; i < array.length; i++) {
      if (array[i] != 2 * i) {
        System.out.printf("%d found at offset %d\n", array[i], i);
        return;      // A single failure crashes the rocket.
      }
    }
  }
}
```

Here are four runs, in which `runBad` fails twice, while `runGood` never fails:

```
$ java ArrayAdding.java
206881 found at offset 206881
Starting: runGood
Starting: runBad
$ java ArrayAdding.java
Starting: runGood
Starting: runBad
$ java ArrayAdding.java
Starting: runGood
Starting: runBad
468 found at offset 468
$ java ArrayAdding.java
Starting: runGood
Starting: runBad
$
```

As per the comment in the code, a system that fails once in a while will eventually kill someone. If you don't believe this, check out the Therac-25, in which a similar race condition killed a few people being treated for cancer.

There are examples of the `synchronized` keyword in several recipes, including Recipe 15.4.

11.5 Simplifying Synchronization with Locks

Problem

You want a more flexible means of synchronizing threads than the `synchronized` keyword.

Solution

Use the `Lock` mechanism in `java.util.concurrent.locks`.

Discussion

Use the java.util.concurrent.locks package; its major interface is `Lock`. This interface has several methods for locking and one for unlocking. Here is the general pattern for using `Lock`:

```
Lock thelock = ....
try  {
        lock.lock( );
        // do the work that is protected by the lock
} finally {
        lock.unlock( );
}
```

The point of putting the `unlock()` call in the `finally` block is, of course, to ensure that it is not bypassed if an exception occurs. (The code may also include one or more `catch` blocks, as required by the work being performed.)

The improvement here, compared with the traditional synchronized methods and blocks, is that using a `Lock` actually looks like a locking operation! And, as I mentioned, several means of locking are available, shown in Table 11-1. The `tryLock()` and `lockInterruptibly()` methods in the `Lock` interface solve another issue with synchronized blocks: that they block a thread in an uninterruptible state while it is waiting for a lock. There is no way to attempt to enter a synchronized block with a timeout governing how long it should wait. This makes deadlock significantly more likely.

Table 11-1. Locking methods of the `Lock` class

Return type	Method	Meaning
void	`lock( )`	Get the lock, even if you have to wait until another thread frees it first
boolean	`tryLock( )`	Get the lock only if it is free right now
boolean	`tryLock(long time, TimeUnit units) throws InterruptedException`	Try to get the lock, but only wait for the length of time indicated
void	`lockInterruptibly( ) throws InterruptedException`	Get the lock, waiting unless interrupted
void	`unlock( )`	Release the lock

The `TimeUnit` class lets you specify the units for the amount of time specified, including `TimeUnit.SECONDS`, `TimeUnit.MILLISECONDS`, `TimeUnit.MICROSECONDS`, and `TimeUnit.NANOSECONDS`.

In all cases, the lock must be released with `unlock()` before it can be locked again.

11.6 Locking with One Writer, Many Readers

Problem

You have many threads that need to read some data, while one thread occasionally needs to write or update the same date.

Solution

Use a readers-writer lock, the `ReentrantReadWriteLock`.

Discussion

The standard `Lock` is useful in many applications, but depending on the application's requirements, other types of locks may be more appropriate. Applications with asymmetric load patterns may benefit from a common pattern called the *reader-writer lock*. I call the lock in this recipe a readers-writer lock to emphasize that there can be many readers but only one writer. It's actually a pair of interconnected locks; any number of readers can hold the read lock and read the data, as long as it's not being written (shared read access). A thread trying to lock the write lock, however, waits until all the readers are finished and then locks them out until the writer is finished (exclusive write access). To support this pattern, both the `ReadWriteLock` interface and the implementing class `ReentrantReadWriteLock` are available. The interface has only two methods, `readLock()` and `writeLock()`, which provide a reference to the

appropriate `Lock` implementation. *These methods do not, in themselves, lock or unlock the locks*; they only provide access to them, so it is common to see code like this:

```
rwlock.readLock( ).lock( );
...
rwlock.readLock( ).unlock( );
```

To demonstrate `ReadWriteLock` in action, I wrote the business logic portion of a web-based voting application. It could be used in voting for candidates or for the more common web poll. Presuming that you display the results on the home page and change the data only when somebody takes the time to click a response to vote, this application fits one of the intended criteria for `ReadWriteLock`—that is, that you have more readers than writers. The main class, `ReadersWritersDemo`, is shown in Example 11-8. The helper class `BallotBox` is online; it simply keeps track of the votes and returns a read-only `Iterator` upon request. Note that in the `run()` method of the reading threads, you could obtain the iterator while holding the lock but release the lock before printing it. This allows greater concurrency and better performance, but it could (depending on your application) require additional locking against concurrent update.

Example 11-8. main/src/main/java/threads/ReadersWriterDemo.java

```
public class ReadersWriterDemo {
  private static final int NUM_READER_THREADS = 3;

  public static void main(String[] args) throws Exception {
    new ReadersWriterDemo().demo();
  }

  /** Set this to true to end the program */
  private volatile boolean done = false;

  /** The data being protected. */
  private final BallotBox theData;

  /** The read lock / write lock combination */
  private final ReadWriteLock lock = new ReentrantReadWriteLock();

  /**
   * Constructor: set up some quasi-random initial data
   */
  public ReadersWriterDemo() throws Exception {
    List<String> choicesList = new ArrayList<>();
    choicesList.add("Agree");
    choicesList.add("Disagree");
    choicesList.add("No opinion");
    theData = new BallotBox(choicesList);
  }
```

```
/**
 * Run a demo with more readers than writers
 */
private void demo() {

  // Start two reader threads
  for (int i = 0; i < NUM_READER_THREADS; i++) {
    Thread.startVirtualThread(() -> {
        while (!done) {
          lock.readLock().lock();
          try {
            theData.forEach(p ->
              System.out.printf("%s: votes %d%n",
                p.getName(),
                p.getVotes()));
          } finally {
            // Unlock in "finally" to be sure it gets done.
            lock.readLock().unlock();
          }

          try {
            Thread.sleep(((long)(Math.random() * 1000)));
          } catch (InterruptedException ex) {
            // nothing to do
          }
        }
    });
  }

  // Start one writer thread to simulate occasional voting
  Thread.startVirtualThread(() -> {
      while (!done) {
        lock.writeLock().lock();
        try {
          theData.voteFor(
            // Vote for random candidate :-)
            // Performance: should have one PRNG per thread.
            (((int)(Math.random() *
              theData.getCandidateCount()))));
        } finally {
          lock.writeLock().unlock();
        }
        try {
          Thread.sleep(((long)(Math.random()*1000)));
        } catch (InterruptedException ex) {
          // nothing to do
        }
      }
  });

  // In the main thread, wait a while then terminate the run.
  try {
```

```
            Thread.sleep(10 * 1000);
        } catch (InterruptedException ex) {
            // nothing to do
        } finally {
            done = true;
        }
    }
}
```

Because this is a simulation and the voting is random, it does not always come out 50/50. In two consecutive runs, the following were the last line of each run:

```
Agree(6), Disagree(6)
Agree(9), Disagree(4)
```

See Also

The `Lock` interface also makes available `Condition` objects, which provide more flexibility. Consult the online documentation for more information.

It's a bit unfortunate that these locks are not `AutoCloseable`. One might wrap them in a subclass or delegation with `close()` delegating to `unlock()`. See this StackOverflow post for more information.

11.7 Sharing Data Among Threads—ThreadLocal and ScopedValue: Structuring Concurrency

Problem

You need to share data among multiple threads.

Solution

Use a `ThreadLocal`, or, in Java 21+ preview, use a `ScopedValue`.

Discussion

In Java, all *nonlocal* data is shared between threads. The `ThreadLocal` class, in Java since 1.3, provides a way to hide data such that there is an instance for each thread, by effectively providing a `Map` from the current thread to an arbitary object, giving the illusion of each thread having its own copy of a given variable. Suppose we have multiple threads, each reading records sequentially from a shared data store. We could write this two ways, using either aggregation or inheritance. The example in Example 11-9 uses inheritance, subclassing `ThreadLocal` to provide an `initialValue()` method; this style is preferred.

Example 11-9. main/src/main/java/threads/ThreadLocalDemo.java

```
/** This ThreadLocal holds the Client reference for each Thread.
 * Make ThreadLocal instance static, to show that it is not an instance variable
 * but is derived from Thread.currentThead() regardless of static/instance access.
 */
private static ThreadLocal<Client> myClient = new ThreadLocal<Client>() {
  // initialValue() is called magically when you first call get().
  @Override
  protected synchronized Client initialValue() {
    return new Client(++clientNum);
  }
};
// IRL this method would do something useful.
@Override
public void run() {
  System.out.println("Thread " + Thread.currentThread().getName() +
    " sees client " + myClient.get());
}

public static void main(String[] args) {
  new ThreadLocalDemo("demo 1").start();
  new ThreadLocalDemo("demo 2").start();
  Thread.yield();
  System.out.println("Main program sees client " + myClient.get());
}

/** A serial number for clients */
private static int clientNum = 0;

/** Simple data class, in real life clients would have more fields! */
private static record Client(int clientNum) { }
```

Because threading is necessarily indeterminate about the order in which code will run, this produces one of several different outputs when run:

```
$ java ThreadLocalDemo.java
Main program sees client Client[clientNum=2]
Thread demo 2 sees client Client[clientNum=3]
Thread demo 1 sees client Client[clientNum=1]
$ r
java ThreadLocalDemo.java
Thread demo 1 sees client Client[clientNum=1]
Main program sees client Client[clientNum=2]
Thread demo 2 sees client Client[clientNum=3]
$
```

If you change the code to use an `AtomicInteger` and remove the `protected` attribute from `initialValue()`, the code becomes fast enough that it appears deterministic—`main()` always wins. This is done in *ThreadLocalDemo2.java* in the same directory.

ThreadLocalDemo3.java demonstrates using explicit creation of the object via aggregation rather than subclassing `ThreadLocal`.

21P Java 21 introduces a preview class called `ScopedValue`. This class provides access to a value in the thread it's associated with, and, importantly, in any method called by that thread. In effect, this allows for the passing of data to a method multiple levels down in the call stack without adding it as parameter(s) to every method along the way.

As a simple example:

```
static final ScopedValue<Client> CLIENT = ScopedValue.newInstance();
private final Client client = new Client().
ScopedValue.runWhere(CLIENT, client, () -> doSomething());
```

Assuming `doSomething()` calls `doSomethingElse()`, which in turn calls `doYetMore()`, the latter can find the client object by calling:

```
Client client = client.GET();
```

A runnable version of this code is in Example 11-10.

Example 11-10. main/src/main/java/threads/ScopedValueDemo1.java

```
static final ScopedValue<Client> CLIENT = ScopedValue.newInstance();

private final Client currentClient = new Client(1234);

void main() {
  ScopedValue.runWhere(CLIENT, currentClient, () -> doSomething());
}

void doSomething() {
  doSomethingElse();
}
void doSomethingElse() {
  doYetMore();
}
void doYetMore() {
  Client client = CLIENT.get();
  System.out.println("In doYetMore, client = " + client);
}

record Client(int id) { }
```

The `ScopedValue` object is bound when the `runWhere` method begins, and unbound when it exits.

Note that a `ScopedValue` is unsurprisingly tied to a given thread. To pass a value to multiple threads you need a `StructuredTaskScope` to bind the value in multiple threads. Such objects must be immutable to prevent race conditions, or an appropriate synchronization means must be used. Suppose we wanted to run three methods each in a separate thread. We simply wrap all the methods in the `StructuredTaskScope`, as shown in Example 11-11.

Example 11-11. main/src/main/java/threads/StructuredTaskScopedemo.java

```
private static final ScopedValue<Client> CLIENT = ScopedValue.newInstance();

private final Client currentClient = new Client(1234);

void main() {

  ScopedValue.runWhere(CLIENT, currentClient, () -> {
    try (var scope = new StructuredTaskScope<Object>()) {
      scope.fork(doSomething);
      scope.fork(doSomethingElse);
      scope.fork(doYetMore);
      scope.join();
    } catch (InterruptedException iex) {
      System.out.println("Uncool! You interrupted me.");
    }
  });
}

Callable<String> doSomething = () -> {
  System.out.println("In doSomething, client " + CLIENT.get());
  return "One";
};
Callable<String> doSomethingElse = () -> {
  System.out.println("In doSomethingElse, not much here.");
  return "Two";
};
Callable<Integer> doYetMore = () -> {
  System.out.println("In doYetMore, using client " + CLIENT.get());
  return 42;
};

record Client(int id) { }
```

The `join()` method waits for completion of all tasks, and the `close()` method abandons the task scope. It's a mistake to call the former without the latter. The scope also has a `shutdown()` method to shut down an entire task scope without closing it; this method *cancels all unfinished subtasks* by interrupting the threads, and will not start any new threads. If the scope's owner is waiting in the `join()` method then it will wake up.

As for when to use `ThreadLocal` versus `ScopedValue`, the documentation states:

> "A `ScopedValue` should be preferred over a `ThreadLocal` for cases where the goal is "one-way transmission" of data without using method parameters. While a `ThreadLocal` can be used to pass data to a method without using method parameters, it does suffer from a number of issues:
>
> - `ThreadLocal` does not prevent code in a faraway callee from setting a new value.
> - A `ThreadLocal` has an unbounded lifetime and thus continues to have a value after a method completes, unless explicitly removed.
> - Inheritance is expensive—the map of thread-locals to values must be copied when creating each child thread."

See Also

The ideas behind structured concurrency are detailed in JEP 428.

11.8 Simplifying Producer/Consumer with the Queue Interface

Problem

You need to control producer/consumer implementations involving multiple threads.

Solution

Use the `Queue` interface or the `BlockingQueue` subinterface.

Discussion

As an example of the simplifications possible with the `java.util.Concurrent` package, consider the standard producer/consumer program. An implementation synchronized using traditional `Thread` code (`wait()` and `notifyAll()`) is in the online source as `ProdConsThreadAPI` (its code is not shown here). Example 11-12 uses the `java.util.BlockingQueue` (a subinterface of `java.util.Queue`) to reimplement `ProdConsThreadAPI` in about two-thirds the number of lines of code, and it's simpler. The application simply puts items into a queue and takes them from it. In the example, I have four producers and only three consumers, so the producers eventually wait. Running the application on one of my older notebooks, the producers' lead over the consumers increases to about 350 over the 10 seconds or so of running it.

Another way of handling producer/consumer is with the `Semaphore` class in `java.util.concurrent`. Semaphores were designed by famous computer scientist Edsger Dijkstra to provide a thread-safe way of maintaining a count of available

resources. This is not expanded on in the book, but there is an implementation online in *main/src/main/java/threads/ProdConsSemaphore.java*.

Example 11-12. main/src/main/java/threads/ProdConsQueues.java

```
public class ProdConsQueues {

  protected volatile boolean done = false;
  protected ExecutorService threadPool =
    Executors.newCachedThreadPool();

  /** Inner class representing the Producer side */
  class Producer implements Runnable {

    protected BlockingQueue<Object> queue;

    Producer(BlockingQueue<Object> theQueue) { this.queue = theQueue; }

    public void run() {
      try {
        while (!done) {
          Object justProduced = getRequestFromNetwork();
          queue.put(justProduced);
          System.out.println(
            "Produced 1 object; List size now " + queue.size());
        }
      } catch (InterruptedException ex) {
        System.out.println("Producer INTERRUPTED");
      }
    }

    Object getRequestFromNetwork() {  // Simulation of reading from client
      try {
          Thread.sleep(10); // simulate time passing during read
      } catch (InterruptedException ex) {
         System.out.println("Producer Read INTERRUPTED");
      }
      return new Object();
    }
  }

  /** Inner class representing the Consumer side */
  class Consumer implements Runnable {
    protected BlockingQueue<Object> queue;

    Consumer(BlockingQueue<Object> theQueue) { this.queue = theQueue; }

    public void run() {
      try {
        while (true) {
          Object obj = queue.take();
```

```
            int len = queue.size();
            System.out.println("List size now " + len);
            process(obj);
            if (done) {
              return;
            }
          }
        } catch (InterruptedException ex) {
            System.out.println("CONSUMER INTERRUPTED");
        }
      }

      void process(Object obj) {
        // Thread.sleep(123) // Simulate time passing
        System.out.println("Consuming object " + obj);
      }
    }

    ProdConsQueues(int nP, int nC) {
      BlockingQueue<Object> myQueue = new LinkedBlockingQueue<>();
      for (int i=0; i<nP; i++)
        threadPool.submit(new Producer(myQueue));
      for (int i=0; i<nC; i++)
        threadPool.submit(new Consumer(myQueue));
    }

    public static void main(String[] args)
      throws IOException, InterruptedException {

      // Start producers and consumers
      int numProducers = 4;
      int numConsumers = 3;
      ProdConsQueues pc = new ProdConsQueues(numProducers, numConsumers);

      // Let the simulation run for, say, 10 seconds
      Thread.sleep(10*1000);

      // End of simulation - shut down gracefully
      pc.done = true;
      pc.threadPool.shutdown();
    }
}
```

`ProdConsQueues` is superior to `ProdConsThreadAPI` in almost all aspects. However, the queue sizes that are output no longer necessarily reflect the size of the queue after the object is inserted or removed. Because there's no longer any locking ensuring atomicity here, any number of queue operations could occur on other threads between the `Producer` thread's `queue.put()` and the `Consumer` thread's queue size query.

11.9 Optimizing Parallel Processing with Fork/Join

Problem

You want to optimize use of multiple processors and/or large problem spaces.

Solution

Use the Fork/Join framework.

Discussion

Fork/Join is an `ExecutorService` intended mainly for reasonably large tasks that can naturally be divided recursively, where you don't have to ensure equal timing for each division. It uses work-stealing to keep threads busy.

The basic means of using Fork/Join is to extend `RecursiveTask` or `RecursiveAction` and override its `compute()` method along these lines:

```
if (assigned portion of work is "small enough") {
        perform the work myself
} else {
        split my work into two pieces
        invoke the two pieces and await the results
}
```

There are two classes: `RecursiveTask` and `RecursiveAction`. The main difference is that `RecursiveTask` has each step of the work returning a value, whereas `RecursiveAction` does not. In other words, the `RecursiveAction` method `compute()` has a return type of `void`, whereas the `RecursiveTask` method of the same name has a return type of `T`, some type parameter. You might use `RecursiveTask` when each call returns a value that represents the computation for its subset of the overall task, in other words, to divide a problem like summarizing data—each task would summarize one part and return that. You might use `RecursiveAction` to operate over a large data structure performing some transform of the data in place.

There are two demos of the Fork/Join framework here, named after the `ForkJoinTask` that each subclasses:

- `RecursiveTaskDemo` uses `fork()` and `join()` directly.
- `RecursiveActionDemo` uses `invokeAll()` to invoke the two subtasks. `invoke()` is just a `fork()` and a `join()`; and `invokeAll()` just does this repeatedly until done. Compare the versions of `compute()` in Examples 11-13 and 11-14 and this will make sense.

Example 11-13. main/src/main/java/threads/RecursiveActionDemo.java

```
/** A trivial demonstration of the "Fork/Join" framework:
 * square a bunch of numbers using RecursiveAction.
 * We use RecursiveAction here b/c we don't need each
 * compute() call to return its result; the work is
 * accumulated in the "dest" array.
 * @see RecursiveTaskDemo when each computation has to return a value.
 */
public class RecursiveActionDemo extends RecursiveAction {

  @Serial
  private static final long serialVersionUID = 3742774374013520116L;

  static int[] raw = {
    19, 3, 0, -1, 57, 24, 65, Integer.MAX_VALUE, 42, 0, 3, 5
  };
  static int[] sorted = null;

  int[] source;
  int[] dest;
  int length;
  int start;
  final static int THRESHOLD = 4;

  public static void main(String[] args) {
    sorted = new int[raw.length];
    RecursiveActionDemo fb =
      new RecursiveActionDemo(raw, 0, raw.length, sorted);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(fb);
    System.out.print('[');
    for (int i : sorted) {
      System.out.print(i + ",");
    }
    System.out.println(']');
    pool.close();
  }

  public RecursiveActionDemo(int[] src, int start, int length, int[] dest) {
      this.source = src;
      this.start = start;
      this.length = length;
      this.dest = dest;
    }

  @Override
  protected void compute() {
    System.out.println("RecursiveActionDemo.compute()");
    if (length <= THRESHOLD) { // Compute Directly
      for (int i = start; i < start + length; i++) {
        dest[i] = source[i] * source[i];
```

```
      }
    } else {              // Divide and Conquer
      int split = length / 2;
      invokeAll(
        new RecursiveActionDemo(source, start,         split,          dest),
        new RecursiveActionDemo(source, start + split, length - split, dest));
    }
  }
}
```

Example 11-14. main/src/main/java/threads/RecursiveTaskDemo.java

```
/**
 * Demonstrate the Fork/Join Framework to average a large array.
 * Running this on a multi-core machine as e.g.,
 * $ time java threads.RecursiveTaskDemo
 * shows that the CPU time is always greater than the elapsed time,
 * indicating that we are making use of multiple cores.
 * That said, it is a somewhat contrived demo.
 *
 * Use RecursiveTask<T> where, as in this example, each call returns
 * a value that represents the computation for its subset of the overall task.
 * @see RecursiveActionDemo when each computation does not return a value,
 * e.g., when each is just working on some section of a large array.
 */
public class RecursiveTaskDemo extends RecursiveTask<Long> {

  @Serial
  private static final long serialVersionUID = 74277437329822011L;

  static final int N = 10000000;
  final static int THRESHOLD = 500;

  int[] data;
  int start, length;

  public static void main(String[] args) {
    int[] source = new int[N];
    loadData(source);
    RecursiveTaskDemo fb = new RecursiveTaskDemo(source, 0, source.length);
    ForkJoinPool pool = new ForkJoinPool();
    long before = System.currentTimeMillis();
    pool.invoke(fb);
    long after = System.currentTimeMillis();
    long total = fb.getRawResult();
    long avg = total / N;
    System.out.println("Average: " + avg);
    System.out.println("Time :" + (after - before) + " mSec");
    pool.close();
  }

  static void loadData(int[] data) {
```

```
    Random r = new Random();
    for (int i = 0; i < data.length; i++) {
      data[i] = r.nextInt();
    }
  }

  public RecursiveTaskDemo(int[] data, int start, int length) {
      this.data = data;
      this.start = start;
      this.length = length;
  }

  @Override
  protected Long compute() {
    if (length <= THRESHOLD) { // Compute directly
      long total = 0;
      for (int i = start; i < start + length; i++) {
        total += data[i];
      }
      return total;
    } else {              // Divide and conquer
      int split = length / 2;
      RecursiveTaskDemo t1 =
        new RecursiveTaskDemo(data, start,         split);
      t1.fork();
      RecursiveTaskDemo t2 =
        new RecursiveTaskDemo(data, start + split, length - split);
      return t2.compute() + t1.join();
    }
  }
}
```

The biggest undefined part is "small enough"; you may have to do some experimentation to see what works well as a chunk size. Or, better yet, write more code using a feedback control system, measuring the system throughput as the parameter is dynamically tweaked up and down, and have the system automatically arrive at the optimal value for that particular computer system and runtime. This is left as an extended exercise for the reader.

The Fork/Join pool is also used in the implementation of alternative concurrency models such as the actor model (as implemented in Akka). It is more general than the recursive task decomposition class of problems. One of its most attractive features is work-stealing, where a task queued for execution on one thread can be taken by another if it is felt that this would result in it being completed sooner.

11.10 Scheduling Tasks: Future Times, Background Saving in an Editor

Problem

You need to schedule something for a fixed time in the future. You need to save the user's work periodically in an interactive program.

Solution

For one-shot future tasks, use the `Timer` service with a `TimerTask` object. For recurring tasks, either use a background thread, or use the `Timer` service and recompute the next time. For more complex tasks, such as running something at high noon every second Thursday, consider using a third-party scheduling library such as Quartz or, in JavaEE/Jakarta, the EJB Timer Service.

Discussion

There are several ways of scheduling things in the future. For one-shot scheduling, you can use the `Timer` service from `java.util`. For recurring tasks, you can use a `Runnable`, which sleeps in a loop.

Here is an example of the `Timer` service in `java.util`. These are the basics of using this API:

1. Create a `Timer` service object.
2. Use it to schedule instances of `TimerTask` with a legacy `Date` object[1] indicating the date and time when the task should be invoked.

The code in Example 11-15 subclasses `TimerTask` as `Item` to perform a simple notification action in the future, based on reading lines with `year-month-day-hour-minute Task`, such as the following:

```
2025 12 25 10 30 Get some sleep.
2025 12 26 01 27 Finish this program
2025 12 25 01 29 Document this program
```

1 It's unfortunate that this API has not been updated to the "new" (in Java 8) date/time API.

Example 11-15. main/src/main/java/threads/ReminderService.java

```
public class ReminderService {

  /** The Timer object */
  Timer timer = new Timer();

  class Item extends TimerTask {
    String message;
    Item(String m) {
      message = m;
    }
    public void run() {
      message(message);
    }
  }

  public static void main(String[] argv) throws Exception {
    new ReminderService().loadReminders();
  }

  private String dfPattern = "yyyy MM dd hh mm ss";
  private SimpleDateFormat formatter = new SimpleDateFormat(dfPattern);

  protected void loadReminders() throws Exception {

    Files.lines(Path.of("ReminderService.txt")).forEach(aLine -> {

      ParsePosition pp = new ParsePosition(0);
      Date date = formatter.parse(aLine, pp);
      String task = aLine.substring(pp.getIndex());
      if (date == null) {
        System.out.println("Invalid date in " + aLine);
        return;
      }
      System.out.println("Date = " + date + "; task = " + task);
      timer.schedule(new Item(task), date);
    });
  }
```

In real life the program would need to run for long periods of time and use a more sophisticated messaging pattern; here we only show the timing scheduling portion.

The code fragment in Example 11-16 creates a background thread to handle background saves, as in most word processors.

Example 11-16. main/src/main/java/threads/AutoSave.java

```
public class AutoSave implements Runnable {
  /** The FileSave interface is implemented by the main class. */
  protected FileSaver model;
```

```
  /** How long to sleep between tries */
  public static final int MINUTES = 5;
  private static final int SECONDS = MINUTES * 60;

  public AutoSave(FileSaver m) {
    Thread.currentThread().setName("AutoSave Thread");
    Thread.currentThread().setDaemon(true);    // so we don't keep the main app
alive
    model = m;
  }

  public void run() {
    while (true) {    // entire run method runs forever.
      try {
        Thread.sleep(SECONDS*1000);
      } catch (InterruptedException e) {
        // do nothing with it
      }
      if (model.wantAutoSave() && model.hasUnsavedChanges())
        model.saveFile(null);
    }
  }

  // Not shown:
  // 1) saveFile() must now be synchronized.
  // 2) method that shuts down main program must be synchronized on *SAME* object
}

/** Local copy of FileSaver interface, for compiling AutoSave demo. */
interface FileSaver {
  /** Load new model from fn; if null, prompt for new fname */
  public void loadFile(String fn);

  /** Ask the model if it wants AutoSave done for it */
  public boolean wantAutoSave();

  /** Ask the model if it has any unsaved changes, don't save otherwise */
  public boolean hasUnsavedChanges();

  /** Save the current model's data in fn.
   * If fn == null, use current fname or prompt for a filename if null.
   */
  public void saveFile(String fn);
}
```

As you can see in the `run()` method, this code sleeps for five minutes (300 seconds), then checks whether it should do anything. If the user has turned autosave off, or hasn't made any changes since the last save, nothing needs to be done. Otherwise, we call the `saveFile()` method in the main program; this saves the data to the current file. It would be smarter to save it to a recovery file of some name, as the better word processors do.

What's not shown is that now all the methods must be synchronized. It's easy to see why if you think about how the save method would work if the user clicked the Save button at the same time that the autosave method called it, or if the user clicked Exit while the file save method had just opened the file for writing. The strategy of saving to a recovery file gets around some of this, but it still needs a great deal of care.

See Also

For details on `java.util.concurrent`, see the documentation accompanying the JDK. For background on JSR-166, see Doug Lea's home page and his JSR-166 page.

A great reference on Java threading is *Java Concurrency in Practice* by Brian Goetz et al. (Addison-Wesley).

Project Loom: Fibers and Continuations produced the virtual threads implementation and, further, aims to promote easier-to-use, lighter-weight concurrency mechanisms.

CHAPTER 12

Data Science and R

12.0 Introduction

Data science is a relatively new discipline that first came to the attention of many with a 2010 article by O'Reilly's Mike Loukides. While there are many definitions in the field, Loukides distills his detailed observation of and participation in data science into this definition:

> A data application acquires its value from the data itself, and creates more data as a result. It's not just an application with data; it's a data product. Data science enables the creation of data products.

One of the main open source ecosystems for data science software is at Apache and includes Hadoop (which includes the Hadoop Distributed File System [HDFS], Hadoop MapReduce,[1] the Ozone object store, and the YARN scheduler), the Cassandra distributed database, and the Spark compute engine. Read the Modules and Related projects sections of the Hadoop page for a current list.

What's interesting here is that a great deal of this infrastructure, which is taken for granted by data scientists, is written in Java and Scala (a JVM language). Much of the rest is written in Python, a language that complements Java. Many users see only the Python side of things and don't realize that Java is behind some of the infrastructure.

1 *Map/Reduce* is a famous producer-consumer algorithm pioneered by Google to handle large data problems. An unspecified number of producers process *map* data—such as from words on a web page to the page's URL—and a single (or a few) reduce process reduces the maps to a manageable form, such as a list of all the pages that contain the given words. Early on, some data scientists went overboard trying to do everything with Map/Reduce; now the pendulum has swung back to using compute engines like Spark.

Data science (DS) problems can involve a lot of setup, so we'll give only one example from traditional DS, using the Spark framework. Spark is written in Scala, so it can be used directly by Java code.

In the rest of the chapter I'll focus on a language called R, which is widely used both in statistics and in data science (and many other sciences; many of the graphs you see in refereed journal articles are prepared with R). Because R is so widely used it is useful to know. Its primary implementation was not written in Java, but in a mixture of C, Fortran, and R itself. But R can be used within Java, and Java can be used within R. I'll talk about several implementations of R and how to select one, and then I'll show techniques for using Java from R and R from Java, as well as using R in a web application.

Some Java-centric alternatives to R are DataFrame libraries such as dflib (*https://dflib.org*) and dataframe-ec (*https://github.com/vmzakharov/data-frame-ec*). The related topic of machine learning for artificial intelligence is treated separately in Chapter 13.

Nothing in this chapter is required to understand the later chapters; it is here as a discussion of one significant area of Java's applicability.

12.1 Using Data in Apache Spark

Problem

You want to process data using Spark.

Solution

Create a `SparkSession`, use its `read()` function to read a `DataSet`, apply operations, and summarize results.

Discussion

Spark supports data science operations on one or more computers. The Apache Spark website reads, in part:

> Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters… [Spark is] the most widely-used engine for scalable computing.

Quoting Databricks, home of much of the original Spark team (viewed in 2022):[2]

> Apache Spark™ has seen immense growth over the past several years, becoming the de-facto data processing and AI engine in enterprises today due to its speed, ease of use, and sophisticated analytics. Spark unifies data and AI by simplifying data preparation at massive scale across various sources, providing a consistent set of APIs for both data engineering and data science workloads, as well as seamless integration with popular AI frameworks and libraries such as TensorFlow, PyTorch, R and SciKit-Learn.

One thing Spark is good for is dealing with lots of data. Example 12-1 reads an Apache-format logfile and finds (and counts) the lines with 200, 404, and 500 responses. The logfile consists of lines like this, with the status code in a later column. We use a regex (see Chapter 4) to find the status code as a word by itself, preceded and followed by at least one nondigit (the `\D` in the regex).

Example 12-1. spark/src/main/java/sparkdemo/LogReader.java

```
/**
 * Read an Apache logfile and summarize it.
 */
public class LogReader {

  public static void main(String[] args) {

    final String logFile = "/var/wildfly/standalone/log/access_log.log"; ❶
    SparkSession spark =
      SparkSession
              .builder()
              .appName("Log Analyzer")
              .config("spark.master", "local")
              .getOrCreate();                                              ❷
    Dataset<String> logData = spark.read().textFile(logFile).cache();      ❸

    long good = logData.filter(                                            ❹
    new FilterFunction<>() {public boolean call(String s) {
          return s.contains(".*\\D200\\D.*");
        }
      }).count();

    long bad = logData.filter(new FilterFunction<>() {
        public boolean call(String s) {
          return s.matches(".*\\D404\\D.*");
        }
      }).count();

    long ugly = logData.filter(new FilterFunction<>() {
        public boolean call(String s) {
```

2 Databricks offers several free ebooks on Spark from its website; it also offers commercial Spark add-ons.

```
          return s.matches(".*\\D500\\D.*");
        }
      }).count();

    System.out.printf(                                                    ❺
      "Successful transfers %d, 404 tries %d, 500 errors %d\n",
      good, bad, ugly);

    spark.stop();
  }
}
```

❶ Set up the filename for the logfile. It probably should come from `args`.

❷ Start up the `SparkSession` object—the runtime.

❸ Tell Spark to read the logfile and keep it in memory (cache).

❹ Define the filters for 200, 404, and 500 errors. They should be able to use lambdas to make the code shorter, but there's an ambiguity between the Java and Scala versions of `FilterFunction`.

❺ Print the results.

We can't run this as a regular Java program, as it depends on the Spark runtime. To make this compile, you need to add the following to a Maven POM file:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.12</artifactId>
    <version>2.4.4</version>
    <scope>provided</scope>
</dependency>
```

Then you should be able to use `mvn package` to get a JAR file packaged.

To run the app, note the use of the `provided` scope, because we will also download the Apache Spark runtime package from the Spark Download page. Unpack the distribution and set the `SPARK_HOME` environment to its root, i.e., the directory created when unpacking the download (you may find a newer version when you run this):

```
SPARK_HOME=~/spark-3.5.1-bin-hadoop3/
```

Then you can use the `run` script that I've provided in the source download (*javasrc/spark*).

Spark is designed for larger-scale computing than what's in this simple example, so its voluminous output dwarfs the output from my basic sample program. Nonetheless, for an approximately 42,000-line file, I did get this result, buried among the logging:

```
Successful transfers 32555, 404 failures 6539, 500 errors 183
```

As mentioned in "Standard input and output streams" on page 393, there are two output streams from a normal program: the standard output (stdout) and the standard error (stderr). Fortunately, Spark puts all the debug chatter on stderr, so we can redirect the stdout to a file, like this:

```
run 2>spark.log
```

Then only the one line of useful output will be printed.

Finally, if Docker is supported and installed on your system, you can get an interactive Spark session by using:

```
docker run -it --rm spark /opt/spark/bin/spark-shell
```

See Also

Spark is a massive subject, but a necessary tool for many data scientists. You can program Spark in Java (obviously), Scala (a JVM language that promotes functional programming), Python, and probably other languages. You can learn more at the Apache Spark website or from the books, videos, and tutorials online.

12.2 Using R Interactively

Problem

You don't know the first thing about R, but you want to.

Solution

R has been around for ages, with its predecessor, S, existing for a decade before that. There are many books and online resources devoted to this language, as well as an official home page. There are many online tutorials; the R Project hosts one. R itself is available in most systems' package managers, and it can be downloaded from the official download site. The name *CRAN* in these URLs stands for Comprehensive R Archive Network, named in a similar fashion to TeX's CTAN and the Perl language's CPAN.

In this example we'll write some data from a Java program and then analyze and graph it using R interactively.

Discussion

This is merely a brief intro to using R interactively. Suffice to say that R is a valuable interactive environment for exploring data. Here are some simple calculations to show the flavor of the language: a chatty startup (so long I had to cut part of it), simple arithmetic, automatic printing of results if not saved, half-decent errors when you make a mistake, and arithmetic on vectors. You might see some similarities to Java's JShell (see Recipe 1.5); both are REPL (read-evaluate-print loop) interfaces. R adds the ability to save your interactive session (called a *workspace*) when exiting the program, so all your data and function definitions are restored next time you start R. A simple interactive session showing a bit of R's syntax might look like this:

```
$ R

R version 4.2.3 (2023-03-15) -- "Shortstop Beagle"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-unknown-openbsd7.5 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

...

> 2 + 2
[1] 4
> x = 2 + 2
> x
[1] 4
> r = 10 20 30 40 50
Error: unexpected numeric constant in "r = 10 20"
> r = c(10,20,30,45,55,67)
> r
[1] 10 20 30 45 55 67
> r+3
[1] 13 23 33 48 58 70
> r / 3
[1]  3.333333  6.666667 10.000000 15.000000 18.333333 22.333333
>quit()
Save workspace image? [y/n/c]: n
$
```

R purists will usually use the *assignment arrow* `<-` in lieu of the `=` sign when assigning. If you like that, go for it.

This short session barely scratches the surface: R offers hundreds of built-in functions, sample datasets, over a thousand add-on packages, built-in help, and much more. For interactive exploration of data, R really is the one to beat.

Some people prefer a GUI frontend to R. There are several:

- R Studio is a GUI for the standard implementation of R.
- Ride is a GUI frontend for Renjin, an alternative implementation of R written in Java.

Now we want to write some data from Java and process it in R (we'll use Java and R together in later recipes in this chapter). In Recipe 5.9 we discussed the `java.util.Random` class and its `nextDouble()` and `nextGaussian()` methods. The `nextDouble()` and related methods try to give a flat distribution between 0 and 1.0, in which each value has an equal chance of being selected. A Gaussian or normal distribution is a bell curve of values from negative infinity to positive infinity, with the majority of the values clustered around zero (0.0). We'll use R's histogramming and graphics functions to examine visually how well they do this:

```
Random r = new Random();
for (int i = 0; i < 10_000; i++) {
    System.out.println("A normal random double is " + r.nextDouble());
    System.out.println("A gaussian random double is " + r.nextGaussian());
```

To illustrate the different distributions, I generated 10,000 numbers each using `nextRandom()` and `nextGaussian()`. The code for this is in *Random4.java* (not shown here) and is a combination of the preceding sample code with code to print just the numbers into two files. I then plotted histograms using R; the R script used to generate the graph is in *javasrc* under *src/main/resources*, but its core is shown in Example 12-2. The `read.table()` function reads a text file into tabular form. The `layout()` says we want two graphs side by side, and the `hist()` call generates a histogram, naturally.

Example 12-2. R commands to generate histograms

```
png("randomness.png")
us <- read.table("normal.txt")[[1]]
ns <- read.table("gaussian.txt")[[1]]

layout(t(c(1,2)), respect=TRUE)

hist(us, main = "Using nextRandom()", nclass = 10,
       xlab = NULL, col = "lightgray", las = 1, font.lab = 3)

hist(ns, main = "Using nextGaussian()", nclass = 16,
       xlab = NULL, col = "lightgray", las = 1, font.lab = 3)
dev.off()
```

The `png()` call tells R which graphics device to use. Others include `X11()` and `Postscript()`. `read.table()` reads data from a text file into a table; the `[1]` gives us just the data column, ignoring the metadata. The `layout()` call says we want two graphics objects displayed side by side. Each `hist()` call draws one of the two histograms. And `dev.off()` closes the output and flushes any writing buffers to the PNG file. The result is shown in Figure 12-1.

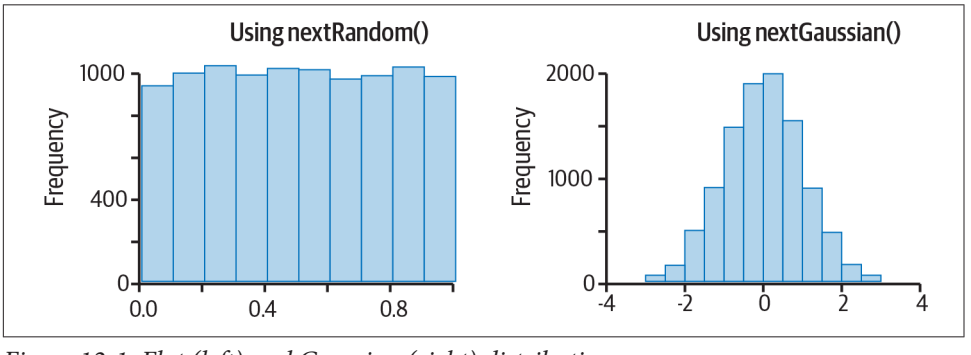


Figure 12-1. Flat (left) and Gaussian (right) distributions

12.3 Comparing/Choosing an R Implementation

Problem

You're not sure which implementation of R to use.

Solution

Look at original R, Renjin, and FastR.

Discussion

Before there was R, there was S, an environment for interactive statistics and programming developed by John Chambers and others at AT&T Bell Labs starting in 1976. I ran into S when supporting the University of Toronto Statistics Department, and again when reviewing a commercial implementation of S, called S-PLUS, for a long-ago glossy magazine called *Sun Expert*. AT&T was only making S source code available to universities and commercial licensees who could not further distribute the code. Frustrated by this policy, two developers at the University of Auckland, Ross Ihaka and Robert Gentleman, developed a clone of S in 1995. They named it R after their own first initials and as a play on the name S. (There is precedent for this: the AWK language popular on Unix/Linux was named for the initials of its designers, Aho, Weinberger, and Kernighan). R grew quickly because it was very compatible with S and was more readily available. The original implementation of R is actively

managed by the R Foundation for Statistical Computing, which also manages the Comprehensive R Archive Network.

Renjin is a fairly complete implementation of R in Java. This project provides built JAR files via their own Maven repository.

FastR is another implementation in Java, designed to run as fast as the original R in the faster Graal VM, that supports direct invocation of JVM code from almost any other programming language. Originally a university project, FastR was shepherded by Oracle as part of `Graal VM`, but is no longer actively maintained by them.

Besides these implementations, R's popularity has led to the development of access libraries for invoking R from many popular programming languages. Rserve is a TCP/IP networked access mode for R, for which Java wrappers exist.

The rest of this chapter covers how to use R from Java, and vice versa.

12.4 Using R from Within a Java App: Renjin

Problem

You want to use Renjin to access R from within a Java application.

Solution

Add Renjin to your Maven or Gradle build, and call it via the Scripting Engine mechanism described in Recipe 18.3.

Discussion

Renjin is a pure-Java, open source reimplementation of R and provides a `ScriptEngine` interface (see Recipe 18.3). Add the following dependency to your build tool:

```
org.renjin:renjin-script-engine:3.5-beta76
```

Of course, by the time you read this, there is probably a later version of Renjin available than the one shown here; use the latest version unless there's a reason not to.

Note that you will also need a `<repository>` entry since the maintainers put their artifacts in the repo at *nexus.betadriven.com* instead of the usual Maven Central. Here's what I used (obtained from the Renjin downloads page):

```
<repositories>
    <repository>
        <id>bedatadriven</id>
        <name>bedatadriven public repo</name>
        <url>https://nexus.bedatadriven.com/content/groups/public/</url>
    </repository>
</repositories>
```

Once that's done, you should be able to access Renjin via the Scripting Engine framework, as in Example 12-3. This runs nicely under an IDE that provides the `CLASSPATH`. To run it under command-line Maven you'd use:

```
mvn exec:java -Dexec.mainClass="otherlang.RenjinScripting"
```

If the application will be used often you could package it, for example with `jpackage` (see Recipe 2.18).

Example 12-3. main/src/main/java/otherlang/RenjinScripting.java

```
/**
 * Demonstrate interacting with the "R" implementation called "Renjin"
 */
public static void main(String[] args) throws ScriptException {
  ScriptEngineManager manager = new ScriptEngineManager();
  ScriptEngine engine = manager.getEngineByName("Renjin");
  engine.put("a", 42);
  Object ret = engine.eval("b <- 2; a*b");
  System.out.println(ret);
}
```

Because R treats all numbers as floating point, like many interpreters, the value printed is `84.0`.

You can also get Renjin to invoke a script file; Example 12-4 invokes the same script used in Recipe 12.2 to generate and plot a batch of pseudorandom numbers.

Example 12-4. main/src/main/java/numbers/Random5.java (Renjin with a script file)

```
private static final String R_SCRIPT_FILE = "/randomnesshistograms.r";
private static final String PLOT_FILE = "randomness.png";
private static int N = 100_000;

public static void main(String[] argv) throws Exception {
  // java.util.Random methods are non-static; we need an instance
  if (argv.length == 1) {
    try {
      N = Integer.parseInt(argv[0]);
    } catch (NumberFormatException ex) {
      System.out.printf("Number %s invalid, using %d\n", argv[0], N);
    }
  }
  System.out.println("Generating " + N + " randoms");
  Random r = new Random();
  double[] us = new double[N], ns = new double[N];
  for (int i=0; i<N; i++) {
    us[i] = r.nextDouble();
    ns[i] =r.nextGaussian();
  }
```

```
    System.out.println("Generating histograms");
    try (InputStream is =
      Random5.class.getResourceAsStream(R_SCRIPT_FILE)) {
      if (is == null) {
        throw new IllegalStateException("Can't open R file ");
      }
      ScriptEngineManager manager = new ScriptEngineManager();
      ScriptEngine engine = manager.getEngineByName("Renjin");
      engine.put("us", us);
      engine.put("ns", ns);
      engine.eval(FileIO.readerToString(new InputStreamReader(is)));
    }
    System.out.println("Done; output in " + PLOT_FILE);
  }
```

Renjin can also be used as a standalone R implementation if you download an all-dependencies JAR file from the Renjin downloads page.

12.5 Using Java from Within an R Session

Problem

You are partway through a computation in standard R and realize that there's a Java library available to do the next step. Or you need to call Java code from within an R session for any reason.

Solution

Install the library rJava, call `.jinit()`, and use `J()` to load classes or invoke methods.

Discussion

Here is the part of an interactive R session in which we install rJava, load its library into R, and invoke `java.time.LocalDate.now()` to get the current date:

```
> install.packages('rJava')                              ❶
trying URL 'https://.../rJava_0.9-11.tgz'
Content type 'application/x-gzip' length 745354 bytes (727 KB)
downloaded 727 KB

The downloaded binary packages are in
    /tmp//Rtmp6XYZ9t/downloaded_packages
> library('rJava')                                       ❷
> J('java.time.LocalDate', 'now')                        ❸
[1] "Java-Object{2025-11-22}"
> d=J('java.time.LocalDate', 'now')$toString()           ❹
> d
[1] "2025-11-22"
```

❶ Install the rJava package; this only needs to be done once. It produces a lot of chatter and may present warnings about missing libraries. Try ignoring those and going on to Step 2.

❷ Load the rJava library, which is needed in every R session.

❸ The `J` function takes one argument of a full class name. If only that argument is given, a class descriptor (like a `java.lang.Class` object) is returned. If more than one argument is given, the second is a static method name, and any subsequent arguments are passed to that method.

❹ Returned objects can have Java methods invoked with the standard R `$` notation; here the `toString()` method is invoked to return just a character string instead of a `LocalDate` object.

The `.jcall` function gives you more control over calling method and return types:

```
> d=J('java.time.LocalDate', 'now')                          ❶
> .jcall(d, "I", 'getYear')                                  ❷
[1] 2025
>
> .jcall("java/lang/System","S","getProperty","user.dir")    ❸
[1] "/home/ian"
> c=J('java/lang/System')                                    ❹
> .jcall(c, "S", 'getProperty', 'user.dir')
[1] "/home/ian"
>
```

❶ Invoke the Java `LocalDate.now()` method and save the result in R variable `d`.

❷ Invoke the Java `getYear()` method on the `LocalDate` object; the `I` tells `.jcall` to expect an integer result.

❸ Call `System.getProperty("user.dir")` and print the result; the `S` tells `.jcall` to expect a string return.

❹ If you will be using a class several times, save the `Class` object and pass it as the first argument of `.jcall()`.

To invoke a constructor, call `.jnew()` with the class name and any arguments:

```
> .jnew('java.math.BigDecimal', 22.2)$doubleValue()
[1] 22.2
>
```

rJava is capable of much more; consult the documentation.

12.6 Using R in a Web App

Problem

You want to display R's data and graphics in a web page on a web server.

Solution

There are several approaches that would achieve this effect:

- Prepare the data, generate graphics as we did in Recipe 12.2, and then incorporate both into a static web page.
- Use one of several R add-on web frameworks, such as shiny or Rook.
- Invoke a JVM implementation of R from within a Servlet, JavaServer Faces (JSF), Spring bean, or other web-tier component.

Discussion

The first approach is trivial and doesn't need discussion here.

For the second, I'll use the R package timevis, which in turn uses shiny. This isn't built into the R library, so we first have to install it, using R's `install.packages()`:

```
$ R
> install.packages('timevis')
> quit()
$
```

This may take a while as it downloads and builds multiple dependencies.

For this demo I have a small dataset with some basic information on medieval literature, which I load and display using shiny and timevis:

```
# Draw the timeline for the epics.

epics = read.table("epics.txt", header=TRUE, fill=TRUE)

# epics

library("timevis")

timevis(epics)
```

When run, this creates a temporary file containing HTML and JavaScript to allow interactive exploration of the data. The library also opens this in a browser, as shown in Figure 12-2. You can explore the data by expanding or contracting the timeline and scrolling sideways.

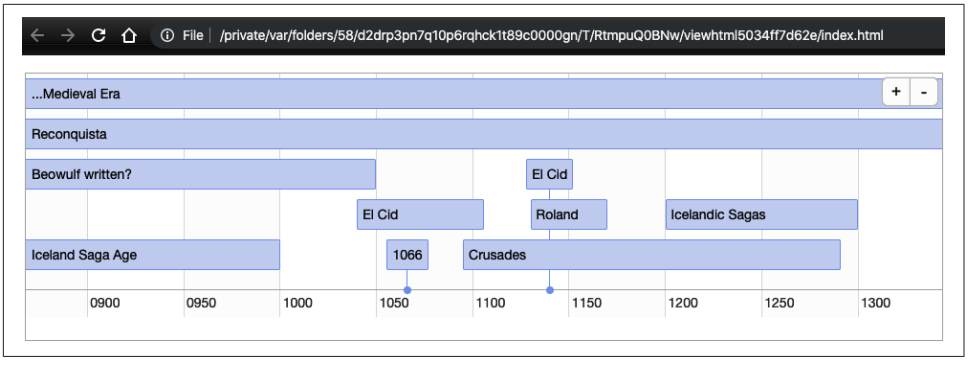


Figure 12-2. Timevis (shiny) in action

Where there are two boxes (El Cid, Icelandic Sagas), the first is when the life or stories took place, and the second is when they were written down.

To expose this on the public web, copy the file (whose full path is displayed in the browser title bar) and the *lib* folder in that same directory into a directory served by the web server. Or just use File→Save As→Complete Web Page within the browser. Either way, you must do this while the R session is running, as the temporary files are deleted when the session ends. Or, if you are familiar with the shiny framework, you can insert the timevis visualization into a shiny application.

CHAPTER 13

Machine Learning/Artificial Intelligence

13.0 Introduction

The notion of "artificial intelligence" was envisioned in science fiction for some years before the first software developments in that direction. Today we still have AI in fiction, like Jarvis, Iron Man's helmet AI. But we also have AI in real life, coming to popular attention starting with ChatGPT in the early 2020s. It's actually been in the works for three quarters of a century.

There are two main approaches to AI: machine learning (ML) and coding. Coding toward AI was one of the factors that led to the creation of the Lisp language in the late 1950s. The goal of coding AI was defined as Artificial General Intelligence, or AGI: an AI that could pass the Turing Test of interacting exactly as a human would. Of course, "intelligence" has many different definitions…

Machine learning (ML), based on neural nets, has been around since at least the 1980s. ML doesn't necessarily aim for AGI, just to perform classification and generation tasks better—or at least faster—than humans. Unlike coded solutions that can be understood by humans, neural nets are based on training. Training data is fed into the AI, which ingests and stores it, for use when interacting with a user. Over time developers connected several neural nets in a row, resulting in the term *deep learning*.

This training process typically consumes massive amounts of computing power, so only small datasets can be trained on a typical mid-2020s desktop computer. Inferencing (using the generated model) takes less overhead, and can be done locally, though rendering (particularly images) is often best done in the cloud. This chapter will focus primarily on the "using" side of things.

One of the original examples of ML is classification. Suppose you want to generate software to tell if a given photographic image contains a human being. The developer

feeds in a large number of examples that have been manually labeled (e.g., "person," "no person") and the software outputs a neural net that "just works"—you don't necessarily know *how* it works, only *how well* it works.

You can think of these classifiers as decision trees. For example, I need to get to a meeting after a night of hanging out late with friends. If it is raining, then if I can find my umbrella I can walk, or if I can find my car keys I can drive, or I must ride the bus or a rideshare. If it's not raining, I can also walk. Figure 13-1 is a simplified example, of course. Some AI software will print out the decision tree it has generated.

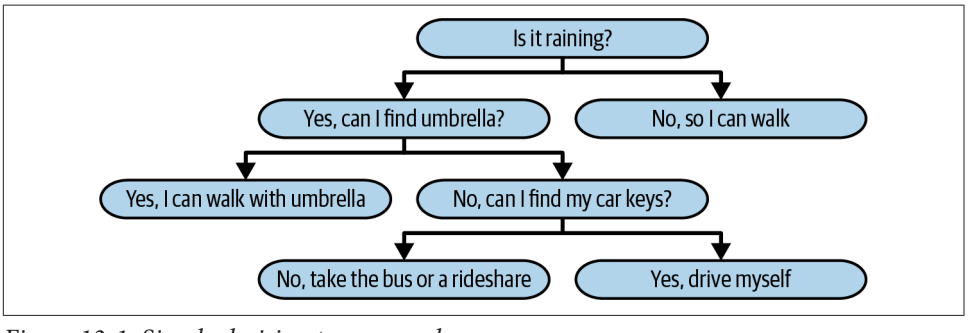

Is it raining?
Yes, can I find umbrella?
No, so I can walk
Yes, I can walk with umbrella
No, can I find my car keys?
No, take the bus or a rideshare
Yes, drive myself

Figure 13-1. Simple decision tree example

Another kind of ML, popularized by (but not exclusive to) ChatGPT, is generative AI. You can feed in all the source code on GitHub and pirated copies of books, and generative AI can tell you how to write any kind of program that was described in its input, and explain how it works, based on its readings. Because these involve massive amounts of data, the generated networks are called large language models (LLMs). As Tim O'Reilly correctly points out in a commentary on the 2023 Biden Administration's AI directive, the use of books and artworks (without consent) presumably violates the copyright on the books, although the source code in public GitHub repos is generally assumed to be fair game. The ancient computing adage GIGO applies as much to AI as to any computation: *garbage in, garbage out*. An AI fed political misinformation may report as "facts" events that never happened, or as "heroes" those who tried to overthrow a legitimate government. Further, some AIs will hallucinate—return a dreamed-up answer—when they don't have a clue. Such errors, of course, can have serious consequences in the real world.

At best, this type of AI can often do tasks better and/or faster than humans, which has the potential to lead to massive job losses in many fields over the coming years. People may stop buying books (like mine!) when they can just ask ChatGPT—though the same fear was raised when Google started indexing everything including GitHub repos. It can be a benefit to some, as in a coding assistant for developers. It can also save lives. Here's one example out of many using classification in medical care. St. Michael's Hospital in Toronto is a regional trauma center, that is, the hospital where

EMTs know to take people banged up in serious car crashes, major accidents, and those injured with malevolent intent (e.g., a crowbar to the head). In addition to examination by a qualified emergency room doctor, incoming brain trauma patients will have brain scans. Patients will partly be triaged by an AI application to see if their injuries require immediate treatment (without which they will shortly die) or whether they will survive if they wait while the more serious injuries are treated. The training for this involves many, many scans that have been tagged with their actual outcomes, along with input from a number of ER doctors with experience in different patient ages, ethnicities, kinds of injuries, etc. It saves lives.

It's sometimes said that machine learning and deep learning have to be done in C++ for efficiency or in Python for the wide availability of software. While these languages have their pros and cons (and their passionate devotees), this is the *Java* Cookbook, so this chapter focuses on a few toolkits for doing ML-type AI in Java. There isn't space for examples of all the good Java-based AI software out there. Some can run on your desktop, some with support for NVIDIA's CUDA API[1] for faster GPU-based processing, so there is no reason to avoid using Java for ML. Table 13-1, lists packages that are currently maintained and have a decent reputation among users.

Table 13-1. Some Java machine learning packages

Library name	Description	Info URL	Source URL
ADAMS	Workflow engine for building/maintaining data-driven, reactive workflows; integration with business processes	*https://adams.cms.waikato.ac.nz*	*https://github.com/waikato-datamining/adams-base*
Deep Java Library	Amazon's ML library	*https://djl.ai*	*https://github.com/awslabs/djl*
Deeplearning4j	DL4J, Eclipse's distributed deep-learning library; integrates w/ Hadoop and Apache Spark	*https://deeplearning4j.org*	*https://github.com/eclipse/deeplearning4j*
ELKI	Data mining toolkit	*https://elki-project.github.io*	*https://github.com/elki-project/elki*
LangChain4j	Unified interface to many popular third-party AI toolkits	*https://docs.langchain4j.dev*	*https://github.com/langchain4j/langchain4j*
Mallet	ML for text processing	*https://mimno.github.io/Mallet/index*	*https://github.com/mimno/Mallet.git*
Spring AI	Unified interface to many popular third-party AI toolkits	*https://spring.io/projects/spring-ai*	*https://github.com/spring-projects/spring-ai*
TensorFlow	Open source framework for creating and deploying ML models	*https://www.tensorflow.org*	*https://github.com/tensorflow/tensorflow*
Tribuo	Unified interface to many popular third-party ML libraries	*https://github.com/oracle/tribuo*	*https://github.com/oracle/tribuo*

1 CUDA originally stood for Compute Unified Device Architecture.

Library name	Description	Info URL	Source URL
Weka	ML algorithms for data mining; tools for data preparation, classification, regression, clustering, association rules mining, and visualization	*https://www.cs.waikato.ac.nz/ml/weka/index.html*	*https://svn.cms.waikato.ac.nz/svn/weka/trunk/weka*

13.1 Some Major AI Software

There are indeed many AI options for Java developers. I've highlighted some of the major ones in the following sections, though most of my examples focus on LangChain4j, which appears to be among the more widely used offerings. You may also refer to Eugen Parschiv's list of Java AI software packages. Also, AI is constantly changing, so you should always check out the online documentation for the latest version of whatever framework you are using.

DeepSeek-R1 was released too close to this book's publication for coverage.

LangChain4j

LangChain4j offers a frontend to numerous API toolkits and datasets. It was self-admittedly somewhat patterned after the Python-based langchain, but shaped with ideas from several other toolkits, and designed specifically as a Java API, not a translation (it doesn't include any Python code). There are examples of LangChain4j in this chapter.

Spring AI

Spring AI offers a frontend to numerous API toolkits and datasets. It can leverage the many other Spring projects, including dependency injection, security, and the like. Spring also offers a convenient tool for building a "starter app," available online at *https://start.spring.io* (be sure to specify Spring AI as a dependency) or interactively via a Spring-provided CLI tool. I don't give examples of Spring AI due to space limitations, but Spring API provides similar functionality to the LangChain4j examples.

TensorFlow

One of the earliest widely used libraries is Google's TensorFlow. Written in Python and C++, TensorFlow can, like most AI toolkits, use GPUs, which provide faster computing for certain types of problems than do the general-purpose CPUs found in AMD/Intel-based computers. A programming language called CUDA is one of sev-

eral for this purpose, and is the one used by TensorFlow. Although not written in Java, TensorFlow has interfaces for Java and other languages. See also the Wikipedia article on TensorFlow.

Oracle Tribuo

Oracle's Tribuo aims to be a comprehensive Java toolkit for AI. As its website introduces it, Tribuo:

> provides tools for classification, regression, clustering, model development, and more. It provides a unified interface to many popular third-party ML libraries like xgboost and liblinear. With interfaces to native code, Tribuo also makes it possible to deploy models trained by Python libraries (e.g. scikit-learn, and pytorch) in a Java program.

Its source code is available in the Tribuo GitHub repository. The name comes from Latin roots meaning to assign or apportion; putatively from the same roots as the English word "tribute."

Other

Although it's not a Java API, Luma Labs makes several interesting online-only applications, including Genie, which creates 3D models from text prompts, and Dream Machine, which creates videos from text.

Ethical Issues in AI

The issue of ethical considerations around AI is the subject of university-level courses, so this will just touch on a few of them.

The first is copyrights. It is widely believed that some of the large data model-based generative AIs may have been trained on works that are copyrighted but used without permission. It's not possible to prove or disprove this idea, because the training data is usually not available (and most people couldn't download it anyway given how massive the datasets are). Obviously, use of copyrighted material for training a generative AI for public use is a violation of copyright, because the information in the copyrighted work will be regurgitated, albeit blended with other similar works. That's just how generative AI operates.

There are solutions; Google researchers have tricked ChatGPT into giving up some of its training data. Of course, this raises privacy issues if information with personally identifiable data was used in training (and it was, according to the paper). Among visual artists, there have been several "image poisoners" who tweak an image so it looks like one thing to an AI but still has its original meaning to humans viewing it as an image.

Another issue is reliability. Consider a medical app that tries to analyze scanned images of neoplasms to see if they are benign (harmless) or malignant (cancerous).

Andrew Tait in the Learning Tree Course "Introduction to AI" compares two variants of one such decision-tree-based classification app. The lighter boxes show the correct diagnoses. The model was trained on thousands of scan images (far more than a human radiologist is trained on). The images were classified manually by trained medical personnel, and corrected with regard to the actual patient outcomes where available.

When tested, one model gives 97% correct classifications, the other only 94%. Is the first one better? One might at first say yes. But look at the "false negative" section—the "more accurate" one will send three patients out of every one hundred home to die, thinking they are cancer-free. The "less accurate" one will only send one patient home oblivious to the cancer destroying their body. It will set five to worrying, but they will probably be OK once they get biopsied. Both are probably about as good as, or better than, a trained radiologist. Which one is better? I'd certainly rather be tested by the second one. Figure 13-2 compares the two models.

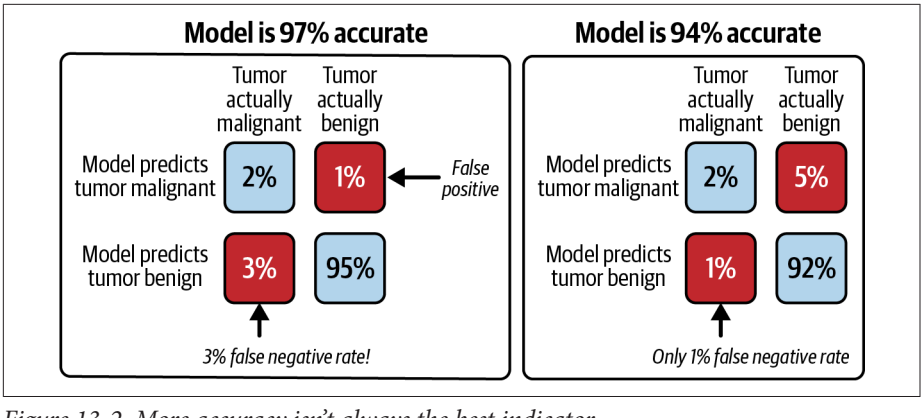


Figure 13-2. More accuracy isn't always the best indicator

AIs are also prone to hallucinations, when they make up answers, or create images of people with the wrong number of fingers or even the wrong number of heads. At the present stage of AI, all of its output needs to be screened by a competent adult human before use.

Yet another issue is bias. Suppose that different ethnic groups in general have different work ethics or financial abilities (this may or may not be true, but assume it is for this discussion). Suppose further that a financial AI model is trained primarily on white middle class people because their data is more readily available to the financial institution developing the model. Is it fair to use this model to make financial decisions (e.g., accept/reject loan applications) about people from any minority background or orientation? I'd say probably not.

Let's not even consider the possibility of massive job losses as AI models get as good as humans in some fields, and are cheaper; the arguments for and against this being likely could fill an entire chapter in a book this size.

There's even a hidden power connection. The electrical energy to power the computers used to train some of the very largest models could probably run a small city for a week; this seems to have implications for both climate activists and poverty activists to ponder. We must leave such issues for others. Further, if millions of people start using AI for spell-checking or web search, they're going to be wasting huge swathes of energy.

One interesting reference on the subject is Mustafa Suleyman's The Coming Wave. Suleyman is the co-founder of DeepMind (which was sold to Google) and Inflection.ai, so he knows a thing or two about AI.

There are many, many ethical questions raised by the increasing use of artificial intelligence. There are, at present, far fewer answers than questions.

See Also

Classification and generative AI have become extremely hot topics in the year or so leading up to this writing. There are predictions of major job losses as AIs continue to improve their ability to perform many tasks better, faster, and cheaper than humans. Once the AIs start rewriting themselves on the fly, all bets are off. Meanwhile, we do what we can with what we have.

The examples in the following recipes use these constants, defined here to ensure they're used consistently:

```
public static final String TEXT_PROMPT =
 "What are the great themes in literature?";
public static final String IMAGE_PROMPT =
 "Charles Darwin in modern Toronto, cartoon style";
```

To run the OpenAI-based models in this chapter, you need to obtain an API key from the OpenAI signup page, and put a few dollars toward the cost of using the examples (most cost under a nickel to run, some under a penny). Those examples also use the method `getOpenAPIKey()`, which retrieves your required OpenAPI key from a text file in your home directory.

13.2 Using ChatGPT Directly

Problem

You want to use ChatGPT directly, without bothering to learn any high-level Java AI toolkits.

Solution

Use the `java.net` API for networking and the `Jackson` API for JSON formatting and parsing.

Discussion

Like almost all online AI tools, ChatGPT can be accessed with a simple REST API using JSON-format requests and responses (REST is explained in Recipe 14.1).

Example 13-1 shows how one might approach this, using the Java API for networking and the Jackson library for XML formatting.

Example 13-1. ai/src/main/java/chatgpt/ChatGptApiDemo.java

```
final static String URL = "https://api.openai.com/v1/chat/completions";
final static boolean DUMP_RAW = false;

public static void main(String[] args) throws Exception {
  System.out.println(chatGPT(Constants.TEXT_PROMPT));
}

public static String chatGPT(String prompt)
    throws IOException, URISyntaxException {
  String apiKey = Constants.getOpenAPIKey();
  String model = "gpt-4o";
  HttpClient client = HttpClient.newHttpClient();

  URI uri = new URI(URL);

  // Send the request
  var request = new ChatGptRequest(model, "user", prompt);
  String json = request.toString();
  System.out.println("Sending this: " + json);

      HttpRequest webRequest = HttpRequest.newBuilder()
      .uri(uri)
      .header("Content-Type", "application/json")
      .header("Authorization", "Bearer " + apiKey)
      .POST(HttpRequest.BodyPublishers.ofString(json))
      .build();
  HttpResponse<String> response = null;
```

```
    try {
      response = client.send(webRequest, HttpResponse.BodyHandlers.ofString());
      System.out.println("Response code: " + response.statusCode());
      System.out.println("Response body: " + response.body());
    } catch (IOException | InterruptedException e) {
      e.printStackTrace();
    }

    // Get a response from ChatGPT
    int n = response.statusCode();
    switch(n) {
      case 429:
        System.out.println("No ChatGPT wants to listen to your chuntering.");
        String rah = null;
        if ((rah =
          response.headers().map().get("retry-after").getFirst()) != null) {
          System.out.println("Try again: " + rah);
        }
        System.out.println("Response body: " + response.body());
        return null;
      case 500: case 501: case 502:
        System.out.println("GPT Server error " + n + " " + response.body());
        return null;
      default:
        System.out.println("HTTP Status was " + n);
    }

    if (DUMP_RAW) {
      System.out.println("Response body: " + response.body());
      return "Answer dumped, no JSON parsing done.";
    } else {
      var mapper = new ObjectMapper();
      // Guard against fields added as GPT evolves
      mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
      ChatGptResponse resp =
        mapper.readValue(response.body(), ChatGptResponse.class);
      System.out.println(resp);
      return resp.choices[0].message.content;
    }
  }

record ChatGptRequest(String model, String role, String prompt) {
  public String toString() {
    return """
      {"model": "%s", "messages": [{"role": "%s", "content": "%s"}]}"""
      .formatted(model, role, prompt);
  }
}

/** These could be records or classes */
static class ChatGptResponse {
  public String id;
```

```
    public ChatGptChoice[] choices;
    public String object;
    public long created;
    public String model;
    public ChatGptUsage usage;
    public String system_fingerprint;

    public String toString() {
      var sb = new StringBuilder();
      for (ChatGptChoice comp : choices) {
        sb.append(comp).append('\n');
      }
      return sb.toString();
    }
  }

  static class ChatGptChoice{
    public int index;
    public ChatGptMessage message;
    public Object logprobs;
    public String finish_reason;
  }

  static class ChatGptMessage {
    public String role;
    public String content;
    public String refusal;
  }

  static class ChatGptUsage {
    public int prompt_tokens;
    public Object prompt_tokens_details;
    public int completion_tokens;
    public Object completion_tokens_details;
    public int total_tokens;
  }
```

`main()` just calls our `chatGPT()` method with the query.

Using an `HttpURLConnection` (see HttpURLConnection example), we connect to the ChatGPT URL, providing an authentication key (for billing!) and setting the content type to JSON.

`ChatGptRequest` is our holder for all the pieces of a request, and will be crudely translated into JSON in its `toString()` method as we write it onto the output side of the connection, which we close to indicate that we're done sending.

ChatGPT uses the particular status code 429 indicating that it's busy or that you have been using too much computing; for that status code we try some error handling.

For a normal status of 200, the code optionally (based on `DUMP_RAW`) prints the results as raw JSON text, or maps it into Java objects defined after main, and extracts just the relevant portion (the message content).

The JSON parsing could be done just using `JSONObject` and `JSONArray`, but whether done that way or not, the response is dependent upon the message format returned by the other end.

13.3 Using ChatGPT via LangChain4j

Problem

You want to get the results of a ChatGPT prompt with minimal effort.

Solution

Use LangChain4j.

Discussion

The use of a higher-level API can greatly simplify coding. LangChain4j is a higher-level API that eases the cost of developing AI-based software, and provides access to a wide range of large language models (LLMs). To start, one need only add these dependencies to *pom.xml* or *build.gradle* (adjusting the version to the latest one available):

```
<properties>
        <langchain4j-version>0.31.0</langchain4j-version>
</properties>

<dependency>
        <groupId>dev.langchain4j</groupId>
        <artifactId>langchain4j</artifactId>
        <version>${langchain4j-version}</version>
</dependency>

<!-- A dependency per provider that you want to use: here, openai -->
<dependency>
        <groupId>dev.langchain4j</groupId>
        <artifactId>langchain4j-open-ai</artifactId>
        <version>${langchain4j-version}</version>
</dependency>

<dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>2.0.13</version>
</dependency>
```

Then, obtaining a query result using the same prompt as used in Recipe 13.2 can be as simple as Example 13-2.

Example 13-2. ai/src/main/java/langchain4j/LangChain4JSimplePrompt.java

```
/**
 * Simple demo of a chat-gpt text prompt and response.
 */
public class LangChain4JSimplePrompt {

    public static void main(String[] args) {

        String apiKey = Constants.getOpenAPIKey();
        ChatLanguageModel model = OpenAiChatModel.withApiKey(apiKey);

        String results = model.generate(Constants.TEXT_PROMPT);

        System.out.println(results);
    }
}
```

When run, this outputs a short outline of the great themes, as requested.

13.4 Making an AI Service with LangChain4j

Problem

You want a higher-level access to AI, such as using an AI to scan through text looking for certain fields.

Solution

Use a service interface returning a POJO containing those fields.

Discussion

For this we define a service interface, which LangChain4j will implement. The example service is to convert plain-text name-and-address information into a structured form, but this mechanism can be used for a wide range of service types. Invoking the method will populate as many of the `Contact` POJO's fields as possible. Example 13-3 shows using this feature to solve the age-old problem of converting plain text with a name, address, etc., into a "contact" information structure with those values as fields, to be imported into whatever contact manager you use. This example also shows a more general method of constructing the "AI" object.

Example 13-3. ai/src/main/java/langchain4j/LangChain4JContactFinder.java

```
public class LangChain4JContactFinder {

  record Contact(                    ❶
    String firstName,
    String lastName,
    String company,
    String street,
    String city,
    String state,
    String code) {

    @Override
    public String toString() {       ❷
      return String.format("""
        Contact {
          name    : %s %s
          company : %s
          street  : %s
          place   : %s, %s %s
        }""",
        firstName, lastName, company,
        street, city, state, code);
    }
  }

  interface ContactFinder {          ❸

    @UserMessage("Extract information about a person from {{it}}")
    Contact extractContactFrom(String text); ❹
  }

  public static void main(String[] args) {

    ChatLanguageModel chatLanguageModel = OpenAiChatModel.builder()
        .apiKey(Constants.getOpenAPIKey())
        .responseFormat("json_object") ❺
        .build();                    ❻

    ContactFinder finder =           ❼
      AiServices.create(ContactFinder.class, chatLanguageModel);

    String[] texts = {               ❽
        "John Doe, 123 Main St, Anytown, AZ 87654",
        "Mary Smith, 456 Cherry Lane, Someplace CA 90210",
        "John Henry Acme Widgets General Delivery Southampton MA",
    };

    for (String text : texts) {      ❾
      Contact ctc = finder.extractContactFrom(text);
      System.out.println(ctc);
```

```
    }
  }
}
```

❶ Nested Record or POJO class will hold the results of the conversion.

❷ `toString()` prints fields in JSON-like format.

❸ The interface to be implemented by the API, with `@UserMessage` telling the API in general what to look for, as if via a direct chat interface.

❹ The method we will call to parse the data into a `Contact` object.

❺ Builder pattern methods tell the API about the model we want to use, and that we want JSON format (which the LangChain4j documentation says works best).

❻ Builder `build()` method actually constructs the extractor object.

❼ Create the service implementation.

❽ Series of demo cases.

❾ For each demo, do the conversion and display the results.

When run, the preceding code prints the three contacts. The first two (not shown) are as expected; the third shows that the AI has done a decent job of separating the parts of the input despite the complete lack of commas. It won't always work, but it does in most common cases.

```
Contact {
        name    : John Henry
        company : Acme Widgets
        street  : General Delivery
        place   : Southampton, MA
}
```

And we are saved from the necessity of creating a JSON request and parsing a JSON object; the completed `Contact` object is returned from the invocation.

13.5 Conversing with Shadows

Problem

Tired of one-way queries, you want to hold an ongoing conversation with an AI, similar to the web mode of ChatGPT.

Solution

Use LangChain4j's `ChatMemory` to retain the context of a conversation.

Discussion

The `MessageWindowChatMemory` is used to retain the context of a conversation. Used in conjunction with a service (as discussed in Recipe 13.4), it remembers the previous interactions with the AI, providing support for an ongoing conversation. In the following demo, we again ask for a list of the great literary themes, but then ask the AI to elaborate on two of them, not in numerical order, and see that the `ChatMemory` has given the AI context to expand on its previous answers. As usual, answers are ruthlessly edited to save paper.

```
>>> What are the great themes in literature?
Some of the great themes in literature include:

1. Love and relationships: This theme explores the complexities of human
relationships, including romantic love, family dynamics, and friendships.
2. Identity and self-discovery...
3. Power and corruption: Literature often explores the abuse of power and
the consequences of corruption in society.
4. War and conflict...
...
10. Freedom and liberation...

>>> Tell me more about the third item
Power and corruption is a recurring theme in literature that explores ...
Literature that explores power and corruption often features characters ...
By examining this theme, literature prompts readers to ...

>>> What more can you say about the first item
Love and relationships is a universal theme in literature that explores...
In literature, love and relationships are often used to explore themes of ...
Romantic love is a common focus in literature, with stories often centering...
Family relationships are also a rich source of material for exploring...
Friendships are an important aspect of the theme of love and relationships...
Overall, the theme of love and relationships in literature serves as...
```

The code that drove the preceding conversation is in Example 13-4. In real life, of course, the questions would be read interactively, either by a human typing or by another AI(!).

Example 13-4. ai/src/main/java/langchain4j/Converse.java

```
interface Chatterer {
  String speak(String response);
}

public class Converse {
```

```
  public static void main(String[] args) {

    String apiKey = Constants.getOpenAPIKey();
    ChatLanguageModel model = OpenAiChatModel.withApiKey(apiKey);

    Chatterer chatter = AiServices.builder(Chatterer.class)
      .chatLanguageModel(model)
      .chatMemory(MessageWindowChatMemory.withMaxMessages(10))
      .build();

    System.out.println(">>> " + Constants.TEXT_PROMPT);
    String list = chatter.speak(Constants.TEXT_PROMPT);
    System.out.println(list);
    System.out.println();

    final String moreOnThird = "Tell me more about the third item";
    System.out.println(">>> " + moreOnThird);
    String third = chatter.speak(moreOnThird);
    System.out.println(third);
    System.out.println();

    final String whosOnFirst = "What more can you say about the first item";
    System.out.println(">>> " + whosOnFirst);
    String first = chatter.speak(whosOnFirst);
    System.out.println(first);
  }
}
```

13.6 Generating Images with LangChain4j

Problem

You want to generate images from prompts quickly and easily.

Solution

Use LangChain4j's `ImageModel`.

Discussion

The setup for generating images is quite simple with LangChain4j, as shown in Example 13-5. ChatGPT does not return images in the REST response, instead providing a URL to view the resulting image. The code extracts this URL and prints it. Then, if running in a desktop environment (e.g., not running in an application server), the program tries to display the resulting image in a browser. We could also read it using Java's `ImageIO`.

Example 13-5. ai/src/main/java/langchain4j/LangChain4JImageMaker.java

```
  public static void main(String[] args) {

    String apiKey = Constants.getOpenAPIKey();
    ImageModel model = OpenAiImageModel.withApiKey(apiKey);
    Response<Image> response = model.generate(Constants.IMAGE_PROMPT);

    var respUrl = response.content().url();

    System.out.printf("Response URL is %s\n", respUrl);

    if (Desktop.isDesktopSupported()) {
      Desktop dtop = Desktop.getDesktop();
      if (dtop.isSupported(Desktop.Action.BROWSE)) {
                try {
                    dtop.browse(URI.create(respUrl.toString()));
                } catch (IOException ex) {
            System.out.println("Unable to open, due to: " + ex);
                }
            } else {
        System.out.println("Desktop Open_URI Action not supported.");
      }
    } else {
      System.out.println("Desktop not supported");
    }
  }
}
```

As with most AI prompts, the results are nondeterministic or nonrepeatable. And sometimes contain hallucinations. Figure 13-3 shows some images that successive runs of this program generated, with the prompt "Charles Darwin in modern-day Toronto." The middle one bears little resemblance to the canonical images of my distant relative Sir Charles, but does feature some Victorian-era buildings alongside modernity. The last picture does look like the great naturalist, but if you blow up the picture, the child in a stroller with a tablet/phone is being pulled by a man with a detached left hand!

Figure 13-3. Generated images

13.7 Mixed Media Prompts: Inferences from Images with LangChain4j

Problem

You want an AI to explain the contents of an image.

Solution

Provide a LangChain4j `UserMessage` prompt containing both the image and a request for what needs explaining.

Discussion

Some AI systems are capable of interpreting the contents of an image. For example, the image in Figure 13-4 might be explained as:

> This image depicts a stylized, comic book-style illustration of a person standing in an urban environment. The person is wearing glasses and casual clothing, including a T-shirt and an open button-down shirt. They appear to be in a modern city setting, with tall skyscrapers and various buildings lining the street. The street is relatively empty, with a bus visible in the background. The sky is clear and blue, contributing to a bright and vibrant atmosphere. The overall aesthetic has a retro comic book feel.

Figure 13-4. An image to explain

To get this description, the `UserMessage` prompt has to provide both the image data and a query such as "Tell me what you see in this image." LangChain4j provides its own `Image` class (not to be confused with the same-named class in java.awt). If the image resides on an external server, you can pass the image's URL. To provide the image directly, just read it into memory, base64 encode it, and pass that. This is illustrated in Example 13-6.

Example 13-6. ai/src/main/java/langchain4j/LangChain4jImageInference.java

```
/**
 * Simple demo of a chat-gpt text+media prompt and response.
 */
public class LangChain4jImageInference {

  public static void main(String[] args) {

    String apikey = Constants.getOpenAPIKey();

    ChatLanguageModel model = OpenAiChatModel.builder()
        .apiKey(apikey)
        .modelName(OpenAiChatModelName.GPT_4_O)
        .maxTokens(100)
        .build();

    var fileName = "mystery-image.png";
    byte[] imageData = null;
    try {
      imageData = Files.readAllBytes(Path.of(fileName));
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
    Image image = Image.builder()
        .mimeType("image/png")
        .base64Data(Base64.getEncoder().encodeToString(imageData))
        .build();
    UserMessage userMessage = UserMessage.from(
        TextContent.from("Explain this image."),
        ImageContent.from(image)
    );

    Response<AiMessage> response = model.generate(userMessage);

    System.out.println(response.content().text());
  }
}
```

13.8 Running AI Locally with ollama

Problem

You wish to run an LLM locally rather than in the cloud, perhaps for reasons of confidentiality.

Solution

One solution is the ollama package, which can easily be installed for the three main operating systems (Linux, macOS, and Microsoft Windows).

Discussion

To run ollama locally, you need to download the package and run it. Ollama will automatically use whatever GPU cores are on your machine; if no suitable GPUs are found, it will "fall back" to running (albeit slowly) on your CPU.

Start by fetching ollama from the project's download site. I installed it on macOS: just unzip the ZIP file into */Applications* or *~/Applications*. I also downloaded the `ollama` shell script to start the app running and interact with it. Then I typed `ollama run mistral`, which caused it to download the Mistral model (the black spot is the terminal-mode progress bar) and give me a prompt:

```
$ ollama run mistral
pulling manifest
pulling ff82381e2bea... 100%  ▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕▕  4.1 GB
pulling (various other bits...)
verifying sha256 digest
writing manifest
removing any unused layers
success
>>> Send a message (/? for help)
>>> What are the great themes in literature?
There are countless themes that can be found in literature, but some of
the most common and universally relevant include:
1. Love and Relationships: ...
>>> /bye
$
```

There are many other LLM's available to be downloaded for use with ollama, listed at *https://ollama.com/library*.

Since my M2 Macintosh has adequate hardware, this started outputting almost immediately.

Of course we want to access this programmatically, so we can embed it into an application instead of running it interactively. LangChain4j has a solution! The toolkit will talk to ollama over a REST connection.

You need to add the langchain4j-ollama library to your *pom.xml* or *build.gradle*:

```
<dependency>
  <groupId>dev.langchain4j</groupId>
  <artifactId>langchain4j-ollama</artifactId>
  <version>${langchain4j-version}</version>
</dependency>
```

Example 13-7 shows accessing the ollama API and the Mistral model via the LangChain4j API.

Example 13-7. ai/src/main/java/langchain4j/LangChain4JOllama.java

```
import dev.langchain4j.model.chat.ChatLanguageModel;
import dev.langchain4j.model.ollama.OllamaChatModel;
import ai.Constants;

class LangChain4JOllama {

    public static void main(String[] args) {

        ChatLanguageModel model = OllamaChatModel.builder()
                .baseUrl("http://127.0.0.1:11434")
                .modelName("mistral")
                .build();

        String answer = model.generate(Constants.TEXT_PROMPT);

        System.out.println(answer);
    }
}
```

Since it's running on the same machine, "localhost" (IPv4 address 127.0.0.1) is the obvious URL hostname. And to be sure it's really running locally, I turned off both WiFi and wired network connections in `System Preferences`, and the program ran as before. The port number 11434 is commonly used by ollama, but if you're not sure, run the ollama program interactively with no arguments, and you'll see some debug chatter with a line similar to this:

```
time=2024-07-04T17:32:58.172-04:00 level=INFO source=routes.go:1111
    msg="Listening on 127.0.0.1:11434 (version 0.1.48)"
```

Running the access via REST appears to take longer, but that's only because the HTTP connection doesn't complete until the AI—which thinks it's talking to a person so it outputs at a human-readable speed—completes. If you want to watch it output, there is a "streaming mode" available in LangChain4j and most similar APIs.

There are many capabilities in the downloadable models; many of the features demonstrated earlier in the chapter (such as image inferencing) may not be available in all models. Either read the documentation for a given model, or just try it out and see if it works. Some models are general purpose; others are specialized, such as `codegemma` for software developers' coding tasks.

See Also

Both LangChain4j and Spring AI are accompanied by a large set of example programs. The LangChain4j examples are at *https://github.com/langchain4j/langchain4j-examples/blob/main/open-ai-examples*. The official Spring AI examples are at *https://github.com/Azure-Samples/spring-ai-azure-workshop*, and a somewhat simplified version of the same material is at Ken Kousen's *https://github.com/kousen/springaiexamples*.

CHAPTER 14

Network Clients

14.0 Introduction

Java can be used to write many types of networked programs. In traditional socket-based code, the programmer is responsible for structuring the interaction between the client and server; the TCP *socket code* simply ensures that whatever data you send gets to the other end. In higher-level types, such as HTTP, RMI, CORBA, and EJB, the software takes over more control. Sockets are often used for connecting to legacy servers; if you were writing a new application from scratch, you'd be better off using a higher-level service.

It may be helpful to compare sockets with the telephone system. Telephones were originally used for analog voice traffic, which is pretty unstructured. Then they began to be used for some layered applications; the first widely popular one was facsimile transmission, or fax. Where would fax have been without the widespread availability of voice telephony? The second wildly popular layered application historically was dial-up TCP/IP. This coexisted with the web to become popular as a mass-market service. Where would dial-up IP be without widely deployed voice lines? And where would the internet be without dial-up IP? Fax and dial-up are mostly gone now, but they paved the way for your smartphone's networked ability, which is what makes it useful (and even seductive as a time sink).

Sockets are layered like that too. The web, RMI, JDBC, CORBA, and EJB are all layered on top of sockets. HTTP is now the most common protocol and should generally be used for new applications when all you want is to get data from point b to point a. Figure 14-1 shows a very simplified diagram of network layers.

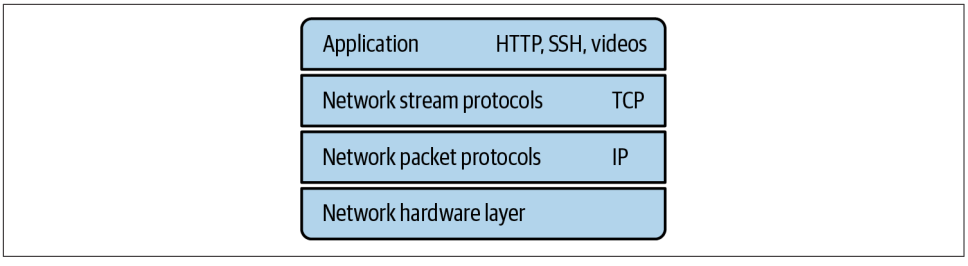


Figure 14-1. Network layers (simplified)

Ever since the alpha release of Java (originally as a sideline to the HotJava browser) in May 1995, Java has been popular as a programming language for building network applications. It's easy to see why, particularly if you've ever built a networked application in C. First, C programmers have to worry about the platform they are on. Unix uses synchronous sockets, which work rather like normal disk files vis-à-vis reading and writing, whereas Microsoft OSes use asynchronous sockets, which use callbacks to notify when a read or write has completed. Java glosses over this distinction. Further, the amount of code needed to set up a socket in C is intimidating. Just for fun, Example 14-1 shows the typical C code for setting up a client socket. Remember, this is only the Unix part. And only the part that makes and closes the connection. To be portable to Windows, it would need some additional conditional code (using C's `#ifdef` mechanism). C's `#include` mechanism requires that exactly the right files be included, and some files have to be listed in a particular order (Java's `import` mechanism is much more flexible).

Example 14-1. main/src/main/java/network/Connect.c (C client setup)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

int
main(int argc, char *argv[])
{
  char* server_name = "localhost";
  struct hostent *host_info;
  int sock;
  struct sockaddr_in server;

  /* Look up the remote host's IP address */
  host_info = gethostbyname(server_name);
  if (host_info == NULL) {
```

```
    fprintf(stderr, "%s: unknown host: %s\n", argv[0], server_name);
    exit(1);
  }

  /* Create the socket */
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("creating client socket");
    exit(2);
  }

  /* Set up the server's socket address */
  server.sin_family = AF_INET;
  memcpy((char *)&server.sin_addr, host_info->h_addr,
            host_info->h_length);
  server.sin_port = htons(80);

  /* Connect to the server */
  if (connect(sock,(struct sockaddr *)&server,sizeof server) < 0) {
    perror("connecting to server");
    exit(4);
  }

  /* Finally, we can read and write on the socket. */
  /* ... */

  (void) close(sock);
}
```

Java replaces all of that setup code (up to the read/write/close) with:

```
    Socket s = new Socket("localhost", 80);
```

In an early recipe, we'll see how to do the connect in this one line of Java (plus a bit of error handling). We'll then cover error handling and transferring data over a socket. Next, we'll take a quick look at a `datagram` or UDP client that implements most of the Trivial File Transfer Protocol (TFTP) that has been used for two decades to boot diskless workstations. We'll end with a program that connects interactively to a chat server.

A common theme through most of these client examples is to use existing servers so that we don't have to generate both the client and the server at the same time. Most of these are services that exist on any standard Unix platform. If you can't find a Unix server near you to try them on, let me suggest that you take an old PC, maybe one that's underpowered for running the latest Microsoft software, and put up a free, open source Unix system on it. My personal favorite is OpenBSD, and the market's overall favorite is Linux. Both are readily available, can be installed for free over the internet, and offer all the standard services used in the client examples, including the time servers and TFTP. Java runs on all of them.

I also provide basic coverage of web services clients. The term *web services* has come to mean program-to-program communication using HTTP. The two general categories are SOAP-based and REST-based. REST services are very simple—you send an HTTP request and get back a response in plain text, JSON (Chapter 16), or XML. SOAP is more complicated and not covered in this book. There is more information on the client-side connections in *Java Network Programming* by Elliotte Rusty Harold (O'Reilly). I don't cover the server-side APIs for building web services—JAX-RS and JAX-WS—because these are covered in several O'Reilly books.

14.1 HTTP/REST Web Client—Modern API 11

Problem

You need to read from a URL, for example, to connect to a RESTful web service or to download a web page or other resource over HTTP/HTTPS.

Solution

Use the standard Java 11 `HttpClient` or the `URLConnection` class.

This technique applies anytime you need to read from a URL, not just when you need to connect to a RESTful web service.

Discussion

Prior to Java 11, you had to either use the `HttpURLConnection` class or download and use the older Apache HttpClient library. With Java 11, there is a fairly easy-to-use and flexible API in standard Java. The `HttpURLConnection` is also still available.

As our basic example, we'll use Google's Suggest service, that is, what you see when you type the first few characters of a search term into the Google web search engine. This Google service supports various output formats. The base URL is the following:

```
https://suggestqueries.google.com/complete/search?client=firefox&q=
```

Append to it the word for which you want suggestions. The `client=firefox` tells it we want a simple JSON format; with `client=chrome` it contains more fields.

To use the Java HTTP Client API, you need an `HttpClient` object, which you get using the Builder pattern. You then create a `Request` object, as in Example 14-2.

Example 14-2. main/src/main/java/netweb/HttpClientDemo—Create client and request

```
// This object would be kept for the life of an application
HttpClient client = HttpClient.newBuilder()
  .followRedirects(Redirect.NORMAL)
```

```
    .version(Version.HTTP_1_1)
    .build();

  // Build the HttpRequest object to "GET" the urlString
  HttpRequest req =
    HttpRequest.newBuilder(URI.create(urlString +
      URLEncoder.encode(input, StandardCharsets.UTF_8)))
    .header("User-Agent", "Ministry of Silly Walks")
    .GET()
    .build();
```

The `HttpRequest` object can be sent using the client to get an `HttpResponse` object, from which you can get the status and/or the body. Sending can be done either synchronously (if you need the results right away) or asynchronously (if you can usefully do something else in the meantime). This example shows sending it both synchronously and asynchronously:

```
    // Send the request - synchronously
    HttpResponse<String> resp =
      client.send(req, BodyHandlers.ofString());

    // Collect the results
    if (resp.statusCode() == 200) {
      String response = resp.body();
      System.out.println(response);
    } else {
      System.out.printf("ERROR: Status %d on request %s\n",
        resp.statusCode(), urlString);
    }

    // Send the request - asynchronously
    client.sendAsync(req, BodyHandlers.ofString())
      .thenApply(HttpResponse::body)
      .thenAccept(System.out::println)
      .join();
```

Here is the output (given the input is set to `"darwin"`); the line has been broken at commas to make it fit on the page:

```
$ java HttpClientDemo.java
["darwin",["darwin thompson","darwin","darwin awards","darwinism",
 "darwin australia","darwin thompson fantasy","darwin barney",
 "darwin theory","darwinai","darwin dormitorio"]]
```

There are times when you might not want to use the HttpClient library. You may be stuck on a very old version of Java. Or you may be familiar with the legacy code and decide that you don't need the features that the new API provides. In such cases you could use the legacy code in `java.net`, since all that's usually needed here is the ability to open and read from a URL. Here is similar code using a `URLConnection` (actually an `HttpURLConnection`):

```
public class RestClientURLDemo {
  public static void main(String[] args) throws Exception {
    HttpURLConnection conn = (HttpURLConnection)
      URI.create(HttpClientDemo.urlString +
        HttpClientDemo.keyword).toURL()
        .openConnection();
    try (BufferedReader is =
      new BufferedReader(
        new InputStreamReader(conn.getInputStream()))) {
      String line;
      while ((line = is.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
}
```

The output should be identical to what the `HttpClient` version produced.

See Also

Don't confuse this `HttpClient` with the older Apache HttpClient library.

You can find more information on REST services (including implementing the server-side components for them) in Bill Burke's *RESTful Java with JAX-RS 2.0* (O'Reilly).

14.2 Contacting a Socket Server

Problem

You need to contact a server using TCP/IP.

Solution

Just create a `java.net.Socket`, passing the hostname and port number into the constructor.

Discussion

There isn't much to this in Java. When creating a socket, you pass in the hostname and the port number. The `java.net.Socket` constructor does the `gethostbyname()` and the `socket()` system call, sets up the server's `sockaddr_in` structure, and executes the `connect()` call. All you have to do is catch the errors, which are subclassed from the familiar `IOException`. Example 14-3 sets up a Java network client and does the absolute bare minimum to replicate the Google Suggest service from

Example 14-2. It uses `try-with-resources` to ensure that the socket and the two streams are closed automatically when we are done.

Example 14-3. main/src/main/java/network/ConnectSimple.java (simple client connection)

```
import java.net.Socket;
import java.io.*;

/* Client with NO error handling */
public class ConnectSimple {

  public static void main(String[] argv) throws Exception {

    try (Socket sock = new Socket("suggestqueries.google.com", 80);
      PrintStream pout = new PrintStream(sock.getOutputStream());
      BufferedReader is =
        new BufferedReader(new InputStreamReader(sock.getInputStream()))) {

      pout.println("GET /complete/search?client=firefox&q=darwin HTTP/1.0\r");
      pout.println("\r");

      String line;
      while ((line = is.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
}
```

This version does no real error reporting, but a version called `ConnectFriendly` does; we'll see this version in Recipe 14.4.

See Also

Java supports other ways of using network applications. You can also open a URL and read from it (see Recipe 14.8). You can write code so that it will run from a URL, when opened in a web browser, or from an application.

14.3 Finding and Reporting Network Addresses

Problem

You want to look up a host's address name or number or get the address at the other end of a network connection.

Solution

Get an `InetAddress` object.

Discussion

The `InetAddress` object represents the internet address of a given computer or host. It has no public constructors; you obtain an `InetAddress` by calling the static `getByName()` method, passing in either a hostname like *darwinsys.com* or a network address as a string, like 1.23.45.67. All the lookup methods in this class can throw the checked `UnknownHostException` (a subclass of `java.io.IOException`), which must be caught or declared on the calling method's definition. None of these methods actually contact the remote host, so they do not throw the other exceptions related to network connections.

The method `getHostAddress()` gives you the numeric IP address (as a string) corresponding to the `InetAddress`. The inverse is `getHostName()`, which reports the name of the `InetAddress`. This can be used to print the address of a host given its name, or vice versa:

```
public class InetAddrDemo {
  public static void main(String[] args) throws IOException {
    String hostName = "darwinsys.com";
    String ipNumber = "8.8.8.8"; // currently a well-known Google DNS server

    // Show getting the InetAddress (looking up a host) by host name
    System.out.println(hostName + "'s address is " +
      InetAddress.getByName(hostName).getHostAddress());

    // Look up a host by address
    System.out.println(ipNumber + "'s name is " +
      InetAddress.getByName(ipNumber).getHostName());

    // Look up my localhost addresss
    final InetAddress localHost = InetAddress.getLocalHost();
    System.out.println("My localhost address is " + localHost);

    // Show getting the InetAddress from an open Socket
    String someServerName = "google.com";
    // assuming there's a web server on the named server:
    try (Socket theSocket = new Socket(someServerName, 80)) {
      InetAddress remote = theSocket.getInetAddress();
      System.out.printf("The InetAddress for %s is %s%n",
        someServerName, remote);
    }
  }
}
```

You can also get an `InetAddress` from a `Socket` by calling its `getInetAddress()` method. You can construct a `Socket` using an `InetAddress` instead of a hostname string. So, to connect to port number `myPortNumber` on the same host as an existing socket, use this:

```
InetAddress remote = theSocket.getInetAddress( );
Socket anotherSocket = new Socket(remote, myPortNumber);
```

Finally, to look up all the addresses associated with a host—a server may be on more than one network—use the static method `getAllByName(host)`, which returns an array of `InetAddress` objects, one for each IP address associated with the given name.

A static method `getLocalHost()` returns an `InetAddress` equivalent to `localhost` or 127.0.0.1. This can be used to connect to a server program running on the same machine as the client.

The number of all possible IPv4 addresses was sufficient back in the day when the research-based ARPANET gave way to the internet. However, once consumers, and devices like smartphones, came along, the number of devices far exceeded the number of addresses. Two solutions exist:

- Network Address Translation (NAT), which gives computers inside your router or firewall addresses drawn from a "private" pool, and translates them into a public address as packets come and go.
- IPv6, the sixth incarnation of the Internet Protocol (IP), which has a 128-bit address space, whose theoretical range exceeds by 22 orders of magnitude the number of smartphones it would take to completely cover all the land on this planet.[1]

If you want IPv6 addresses, you can use `Inet6Address` instead. There is a `getAllByName()` in `InetAddress` and its two subclasses, `Inet4Address` and `Inet6Address`. Note that if your computer is configured for both, `getAllByName()` might return both IPv4 and IPv6 addresses:

```
$ jshell
> import java.net.*;
> Inet6Address.getAllByName("google.com");
$1 ==> InetAddress[2] { google.com/142.250.190.110,
        google.com/2607:f8b0:4009:809:0:0:0:200e }
```

1 Maximum possible IPv6 addresses = $2^{128} = 3.402824 \times 10^{38}$. Amount of dry land is approximately 148.94 million km^2 or 148.94×10^{12} m^2. Area of one smartphone (Pixel 6A) about 0.01092836 m^2. Divide the total area by the area of one unit: Number of smartphones$\approx 148.94 \times 10^{12} / 0.01092836 \approx 1.363 \times 10^{16}$. Approximate ratio: $10^{38}/10^{16} = 10^{22}$, or 22 orders of magnitude.

See Also

See `NetworkInterface` in Recipe 15.2, which lets you find out more about the networking of the machine you are running on. There is no way to look up services in the standard API yet—that is, to find out that the HTTP service is on port 80. Full implementations of TCP/IP have always included an additional set of resolvers; in C, the call `getservbyname("http", "tcp");` would look up the given service[2] and return a `servent` (service entry) structure whose `s_port` member would contain the value 80. The numbers of established services do not change, but when services are new or installed in nonroutine ways, it is convenient to be able to change the service number for all programs on a machine or network (regardless of programming language) just by changing the services definitions. Java should provide this capability in a future release.

14.4 Handling Network Errors

Problem

There are many things that can go wrong during a network interaction. You want more detailed reporting than just `IOException` if something goes wrong.

Solution

Catch a greater variety of exception classes, and report them separately.

Discussion

`SocketException` has several subclasses; the most notable are `ConnectException` and `NoRouteToHostException`. The names are mostly self-explanatory: the first means that the connection was refused by the machine at the other end (the server machine has no service running on the given TCP port), and the second completely explains the failure. Example 14-4 is an excerpt from the `Connect` program, enhanced to handle these conditions.

Example 14-4. ConnectFriendly.java

```
public class ConnectFriendly {
  public static void main(String[] argv) {
```

2 The location where it is looked up varies. It might be in a file named */etc/services* on Unix; in the *services* file in a subdirectory like *\Windows\System32\drivers\etc* in Microsoft Windows; in a centralized registry such as the Domain Name Service, or Sun's Network Information Services (NIS, formerly YP); or in some other platform- or network-dependent location.

```
    String serverName = argv.length == 1 ? argv[0] : "localhost";
    int tcpPort = 80;
    try (Socket sock = new Socket(serverName, tcpPort)) {

      /* If we get here, we can read and write on the socket. */
      System.out.println(" *** Connected to " + serverName  + " ***");

      /* Do some I/O here... */

    } catch (UnknownHostException e) {
      System.err.println(serverName + " Unknown host");
      System.exit(1);
    } catch (NoRouteToHostException e) {
      System.err.println(serverName + " Unreachable" );
      System.exit(1);
    } catch (ConnectException e) {
      System.err.println(serverName + " Connection Refused");
      System.exit(1);
    } catch (java.io.IOException e) {
      System.err.println(serverName + ' ' + e);
      System.exit(1);
    }
  }
}
```

Some of the messages can be easily demonstrated:

```
$ java main/src/main/java/network/ConnectFriendly.java
 *** Connected to localhost ***
$ java main/src/main/java/network/ConnectFriendly.java darwinsys.com
 *** Connected to darwinsys.com ***
$ java main/src/main/java/network/ConnectFriendly.java adventure.plugh
adventure.plugh Unknown host
$ java main/src/main/java/network/ConnectFriendly.java 10.11.12.13
^C  # Interrupted as it hung for a long time.
$
```

14.5 Reading and Writing Textual Data

Problem

Having connected, you want to transfer textual data.

Solution

Construct a `BufferedReader` or `PrintWriter` from the socket's `getInputStream()` or `getOutputStream()`.

Discussion

The `Socket` class has methods that allow you to get an `InputStream` or `OutputStream` to read from or write to the socket. It has no method to fetch a `Reader` or `Writer`, partly because some network services are limited to ASCII, but mainly because the `Socket` class was decided on before there were `Reader` and `Writer` classes. You can always create a `Reader` from an `InputStream` or a `Writer` from an `OutputStream` using the conversion classes. This is the paradigm for the two most common forms:

```
BufferedReader is = new BufferedReader(
    new InputStreamReader(sock.getInputStream( )));
PrintWriter os = new PrintWriter(sock.getOutputStream( ), true);
```

Example 14-5 reads a line of text from the daytime service, which is offered by full-fledged TCP/IP suites (such as those included with most Unixes). You don't have to send anything to the `Daytime` server; you simply connect and read one line. The server writes one line containing the date and time and then closes the connection. Running it looks like the following code. I started by getting the current date and time on the local host, then ran the `DaytimeText` program to see the date and time on one of my Unix servers:

```
C:\javasrc\network>date
Current date is Sun 01-23-2000
Enter new date (mm-dd-yy):
C:\javasrc\network>time
Current time is  1:13:18.70p
Enter new time:
C:\javasrc\network>java network.DaytimeText myserver
Time on myserver is Sun Jan 26 13:14:34 2025
```

The code is in Example 14-5.

Example 14-5. main/src/main/java/network/DaytimeText.java

```
public class DaytimeText {
  public static final short TIME_PORT = 13;

  public static void main(String[] argv) {
    String server_name = argv.length == 1 ? argv[0] : "localhost";

    try (Socket sock = new Socket(server_name,TIME_PORT);
      BufferedReader is = new BufferedReader(new
        InputStreamReader(sock.getInputStream()));) {
      String remoteTime = is.readLine();
      System.out.println("Time on " + server_name + " is " + remoteTime);
    } catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

The second example, shown in Example 14-6, shows both reading and writing on the same socket. The `Echo` server simply echoes back whatever lines of text you send it. It's not a very clever server, but it is a useful one. It helps in network testing and also in testing clients of this type!

The `converse()` method holds a short conversation with the `Echo` server on the named host; if no host is named, it tries to contact `localhost`, a universal alias[3] for the machine the program is running on.

Example 14-6. main/src/main/java/network/EchoClientOneLine.java

```
public class EchoClientOneLine {
  /** What we send across the net */
  String mesg = "Hello across the net";

  public static void main(String[] argv) {
    if (argv.length == 0)
      new EchoClientOneLine().converse("localhost");
    else
      new EchoClientOneLine().converse(argv[0]);
  }

  /** Hold one conversation across the net */
  protected void converse(String hostName) {
    try (Socket sock = new Socket(hostName, 7);) { // echo server.
      BufferedReader is = new BufferedReader(new
        InputStreamReader(sock.getInputStream()));
      PrintWriter os = new PrintWriter(sock.getOutputStream(), true);
      // Do the CRLF ourself since println appends only a \r on
      // platforms where that is the native line ending.
      os.print(mesg + "\r\n"); os.flush();
      String reply = is.readLine();
      System.out.println("Sent \"" + mesg  + "\"");
      System.out.println("Got  \"" + reply + "\"");
    } catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

It might be a good exercise to isolate the reading and writing code from this method into a `NetWriter` class, possibly subclassing `PrintWriter` and adding the `\r\n` and the flushing.

3 It used to be universal, when most networked systems were administered by full-time system admins who had been trained or served an apprenticeship. Today many machines on the internet don't have `localhost` configured properly.

14.6 Reading and Writing Binary or Serialized Data

Problem

Having connected, you want to transfer binary data, either raw binary data or serialized Java objects.

Solution

For plain binary data, construct a `DataInputStream` or `DataOutputStream` from the socket's `getInputStream()` or `getOutputStream()`. For serialized Java object data, construct an `ObjectInputStream` or `ObjectOutputStream`.

Discussion

The simplest paradigm for reading/writing on a socket is this:

```
DataInputStream is = new DataInputStream(sock.getInputStream());
DataOutputStream is = new DataOutputStream(sock.getOutputStream( ));
```

If the volume of data might be large, insert a buffered stream for efficiency. The paradigm is this:

```
DataInputStream is = new DataInputStream(
    new BufferedInputStream(sock.getInputStream( )));
DataOutputStream is = new DataOutputStream(
    new BufferedOutputStream(sock.getOutputStream( )));
```

The program in Example 14-7 uses another standard service that gives out the time, this one as a binary integer representing the number of seconds since 1900. Because the Java `Date` class base (like Unix time) is 1970, we convert the time base by subtracting the difference between 1970 and 1900. When I used this exercise in a course, most of the students wanted to *add* this time difference, reasoning that 1970 is later. But if you think clearly, you'll see that there are fewer seconds between 1999 and 1970 than there are between 1999 and 1900, so subtraction gives the correct number of seconds. And because the `Date` constructor needs milliseconds, we multiply the number of seconds by 1,000.

The time difference is the number of years multiplied by 365, plus the number of leap days between the two dates (in the years 1904, 1908, . . . , 1968)—19 days.

The integer that we read from the server is a C-language `unsigned int`. But Java doesn't provide an unsigned integer type; normally when you need an unsigned number, you use the next-larger integer type, which would be `long`. But Java also doesn't give us a method to read an unsigned integer from a data stream. The `DataInputStream` method `readInt()` reads Java-style signed integers. There are `readUnsignedByte()` methods and `readUnsignedShort()` methods, but no `readUnsignedInt()`

method. Accordingly, we synthesize the ability to read an unsigned `int` (which must be stored in a `long`, or else you'd lose the signed bit and be back where you started from) by reading unsigned bytes and reassembling them using Java's bit-shifting operators. At the end of the code, we use the new date/time API (see Chapter 6) to construct and print a `LocalDateTime` object to show the current date and time on the local (client) machine, taking the time zone difference from GMT into account:

```
$ java TimeClient.java  localhost
Remote time is 3930648456
BASE_DIFF is 2208988800
Time diff == 1721659656
Time on localhost is 2024-07-22T10:47:36
Local date/time = 2024-07-22T10:47:36.848233148
$ java TimeClient.java darwinsys.com
Remote time is 3930648485
BASE_DIFF is 2208988800
Time diff == 1721659685
Time on darwinsys.com is 2024-07-22T10:48:05
Local date/time = 2024-07-22T10:48:05.438931478
```

Looking at the output, you can see that the server and localhost agree on the time, within a second. That confirms the date calculation code in Example 14-7.

Example 14-7. main/src/main/java/network/TimeClient.java

```
public class TimeClient {
  /** The TCP port for the binary time service. */
  public static final short TIME_PORT = 37;

  /** Seconds between 1970, the time base for dates and times
   * Factors in leap years (up to 2100), hours, minutes, and seconds.
   * Subtract 1 day for 1900, add in 1/2 day for 1969/1970.
   */
  protected static final long BASE_DAYS =
    (long)((1970-1900)*365 + (1970-1900-1)/4);

  /* Seconds since 1970 */
  public static final long BASE_DIFF = (BASE_DAYS * 24 * 60 * 60);

  public static void main(String[] argv) {
    String hostName;
    if (argv.length == 0)
      hostName = "localhost";
    else
      hostName = argv[0];

    try (Socket sock = new Socket(hostName,TIME_PORT);) {
      DataInputStream is = new DataInputStream(new
        BufferedInputStream(sock.getInputStream()));
      // Read 4 bytes from the network, unsigned.
      // Do it yourself; there is no readUnsignedInt().
```

```
        // Long is 8 bytes on Java, but we are using the
        // existing time protocol, which uses 4-byte ints.
        long remoteTime = (
          ((long)(is.readUnsignedByte()) << 24) |
          ((long)(is.readUnsignedByte()) << 16) |
          ((long)(is.readUnsignedByte()) <<  8) |
          ((long)(is.readUnsignedByte()) <<  0));
        System.out.println("Remote time is " + remoteTime);
        System.out.println("BASE_DIFF is " + BASE_DIFF);
        System.out.println("Time diff == " + (remoteTime - BASE_DIFF));
        Instant time = Instant.ofEpochSecond(remoteTime - BASE_DIFF);
        LocalDateTime d = LocalDateTime.ofInstant(time, ZoneId.systemDefault());
        System.out.println("Time on " + hostName + " is " + d.toString());
        System.out.println("Local date/time = " + LocalDateTime.now());
      } catch (IOException e) {
        System.err.println(e);
      }
    }
}
```

Object serialization is the ability to convert in-memory objects to an external form that can be sent serially (as a byte stream). To read or write Java objects via serialization, you need only construct an `ObjectInputStream` or `ObjectOutputStream` from an `InputStream` or `OutputStream`; in this case, the socket's `getInputStream()` or `getOutputStream()`.

This program (and its server) provides a service that isn't a standard part of the TCP/IP stack; it's a service I made up as a demo. The server for this service is introduced in Recipe 15.3. The client code in Example 14-8 is quite similar to the `Daytime Binary` program in the previous recipe, but the server sends us a `LocalDateTime` object already constructed. Example 14-8 shows the portion of the client code that differs from Example 14-7.

Example 14-8. main/src/main/java/network/DaytimeObject.java

```
    try (Socket sock = new Socket(hostName, TIME_PORT);) {
      ObjectInputStream is = new ObjectInputStream(new
        BufferedInputStream(sock.getInputStream()));

      // Read and validate the Object
      Object o = is.readObject();
      if (o == null) {
        System.err.println("Read null from server!");
      } else if ((o instanceof LocalDateTime d)) {
        System.out.println("Time on " + hostName + " is " + d);
      } else {
        throw new IllegalArgumentException(
          "Wanted LocalDateTime, got %s, a %s".formatted(
```

```
            o, o.getClass()));
    }
```

I ask the operating system for the date and time, and then I run the program, which prints the date and time on a networked machine named aragorn:

```
$ date
Thu Dec 26 09:29:02 EST 2025
C:\javasrc\network>java network.DaytimeObject aragorn
Time on aragorn is 2025-12-26T09:29:05.227397
C:\javasrc\network>
```

Again, the results agree within a few seconds.

14.7 Postcards of the Internet: Using UDP Datagrams

Problem

You need to use a datagram connection (UDP) instead of a stream connection (TCP).

Solution

Use `DatagramSocket` and `DatagramPacket`.

Discussion

Datagram network traffic is a kindred spirit to the underlying packet-based Ethernet and IP (Internet Protocol) layers. Unlike a stream-based connection such as TCP, datagram transports like UDP transmit each *packet*, or chunk of data, as a single entity with no necessary relation to any other.[4] A common analogy is that TCP is like talking on the telephone, whereas UDP is like sending postcards or maybe fax messages.

The differences show up most in error handling. Packets can, like postcards, go astray. When was the last time the postman rang your bell to tell you that the post office had lost one of several postcards it was supposed to deliver to you? That's not going to happen, because the post office doesn't keep track of postcards. On the other hand, when you're talking on the phone and there's a noise burst—like somebody yelling in the room, or even a signal dropout—you notice the failure in real time, and you can ask the person at the other end to repeat what they just said.

With a stream-based connection like a TCP socket, the network transport layer handles errors for you: it asks the other end to retransmit. With a datagram transport

4 The UDP packet may need to be fragmented by some networks, but this is not germane to us at the UDP level, because it will reassemble the network packets into our single-entity UDP packet at the other end.

such as UDP, you have to handle retransmission yourself. It's kind of like numbering the postcards you send so that you can go back and resend any that don't arrive—a good excuse to return to your vacation spot, perhaps.

Another difference is that datagram transmission preserves message boundaries. That is, if you write 20 bytes and then write 10 bytes when using TCP, the program reading from the other end will not know if you wrote one chunk of 30 bytes, two chunks of 15, or even 30 individual characters. With a `DatagramSocket`, you construct a `DatagramPacket` object for each buffer, and its contents are sent as a *single* entity over the network; its contents will not be mixed together with the contents of any other buffer. The `DatagramPacket` object has methods like `getLength()` and `setPort()`.

Ian's Basic Steps: UDP Client

UDP is a bit more involved, so I'll list the basic steps for generating a UDP client:

1. Create a `DatagramSocket` with no arguments (the form that takes two arguments is used on the server).
2. Optionally `connect()` the socket to an `InetAddress` (see Recipe 14.3) and port number.
3. Create one or more `DatagramPacket` objects; these are wrappers around a byte array that contains data you want to send and is filled in with data you receive.
4. If you did not `connect()` the socket, provide the `InetAddress` and port when constructing the `DatagramPacket`.
5. Set the packet's length and use `sock.send(packet)` to send data to the server.
6. Use `sock.receive()` to retrieve data.

Note that the optional `connect()` call only establishes the default address for packets; UDB is still a "connectionless" protocol.

So why would we even use UDP? UDP has a lot less overhead than TCP, which can be particularly valuable when sending huge amounts of data over a reliable local network or a few hops on the internet. Over long-haul networks, TCP is generally preferred because TCP handles retransmission of lost packets for you. And obviously, if preserving record boundaries makes your life easier, that may be a reason for considering UDP. UDP is also the way to perform Multicast (send to many receivers simultaneously), though Multicast is out of scope for this discussion.

Example 14-9 is a short program that connects via UDP to the `Daytime` date and time server used in Recipe 14.5. Because UDP has no real notion of connection, the client typically initiates the conversation, which sometimes means sending an empty

packet; the UDP server uses the address information it gets from that to return its response.

Example 14-9. main/src/main/java/network/DaytimeUDP.java

```
public class DaytimeUDP {
  /** The UDP port number */
  public final static int DAYTIME_PORT = 13;

  /** A buffer plenty big enough for the date string */
  protected final static int PACKET_SIZE = 100;

  /** The main program that drives this network client.
   * @param argv[0] hostname, running daytime/udp server
   */
  public static void main(String[] argv) throws IOException {
    if (argv.length < 1) {
      System.err.println("usage: java DayTimeUDP host");
      System.exit(1);
    }
    String host = argv[0];
    InetAddress servAddr = InetAddress.getByName(host);
    DatagramSocket sock = new DatagramSocket();
    byte[] buffer = new byte[PACKET_SIZE];

    // The UDP packet we will send and receive
    DatagramPacket packet = new DatagramPacket(
      buffer, PACKET_SIZE, servAddr, DAYTIME_PORT);

    /* Send empty max-length (-1 for null byte) packet to server */
    packet.setLength(PACKET_SIZE-1);
    sock.send(packet);
    System.out.println("Sent request");

    // Receive a packet and print it.
    sock.receive(packet);
    System.out.println("Got packet of size " + packet.getLength());
    System.out.print("Date on " + host + " is " +
      new String(buffer, 0, packet.getLength()));

    sock.close();
  }
}
```

I'll run it to my Unix box just to be sure that it works:

```
$
$ java network.DaytimeUDP aragorn
Sent request
Got packet of size 26
Date on aragorn is Fri Dec 12 09:29:05 2025
$
```

14.8 URI, URL, or URN?

Problem

Having heard these terms, you want to know the difference between a URI, URL, and URN.

Solution

Read on. Or see the Javadoc for *java.net.URI*.

Discussion

A URL is the traditional name for a network address consisting of a scheme (like `https`) and an address (site name) and resource or pathname. But there are three distinct terms in all:

- URI (Uniform Resource Identifier)
- URL (Uniform Resource Locator)
- URN (Uniform Resource Name)

A discussion near the end of the Java documentation for the `URI` class explains the relationship among URI, URL, and URN. URIs form the set of all identifiers. URLs and URNs are subsets. URIs are the most general; a URI is parsed for basic syntax without regard to the scheme, if any, that it specifies, and it need not refer to a particular server. A URL includes a hostname, scheme, and other components; the string is parsed according to rules for its scheme. When you construct a URL, an `InputStream` is created automatically. URNs name resources but do not explain how to locate them; typical examples of URNs that you will have seen include `mailto:` and occasionally `tel:` references.

The main operations provided by the `URI` class are normalization (removing extraneous path segments including `..`) and relativization (this should be called "making relative," but somebody wanted a single word to make a method name). A `URI` object does not have any methods for opening the URI; for that, you would normally use a string representation of the URI to construct a URL object, like so:

```
URL x = new URL(theURI.toString( ));
```

The program in Example 14-10 shows examples of normalizating, making relative, and constructing a URL from a URI.

Example 14-10. main/src/main/java/netweb/URIDemo.java

```
public class URIDemo {
  public static void main(String[] args)
  throws URISyntaxException, MalformedURLException {

    // Start with un-normalized example
    URI u = new URI("https://darwinsys.com/java/../openbsd/../index.html");
    System.out.println("Raw: " + u);
    URI normalized = u.normalize();
    System.out.println("Normalized: " + normalized);
    final URI BASE = new URI("https://darwinsys.com");
    System.out.println("Relativized to " + BASE + ": " + BASE.relativize(u));

    // A URL is a type of URI
    URL url = URI.create(normalized.toString()).toURL();
    System.out.println("URL: " + url);

    // Demo of non-URL but valid URI
    URI uri = new URI("bean:WonderBean");
    System.out.println(uri);
  }
}
```

Running it produces this output:

```
$ java URIDemo.java
Raw: https://darwinsys.com/java/../openbsd/../index.html
Normalized: https://darwinsys.com/index.html
Relativized to https://darwinsys.com: index.html
URL: https://darwinsys.com/index.html
bean:WonderBean
$
```

14.9 Program: Sockets-Based Chat Client

This program is a very simple chat program. You can't talk to Google Chat, Instagram, or X/Twitter (never mind ancient chat-only protocols like ICQ or AIM) with it, because they each use their own protocol.[5] Rather, this program simply writes to and reads from a server. The server for this will be presented in Chapter 15. How does it look when you run it? Figure 14-2 shows me chatting all by myself one day.

The code is reasonably self-explanatory. We read from the remote server in a thread to make the input and the output run without blocking each other; this is discussed in Chapter 11. The techniques for reading and writing have been discussed through-

5 For an open source program that does provide an IM service to let you talk to various chats from a single program, check out Pidgin or Jabber.

out this chapter. The code is not shown here but is in *javasrc* at *main/src/main/java/chat/ChatClient.java* and online at my GitHub repository.

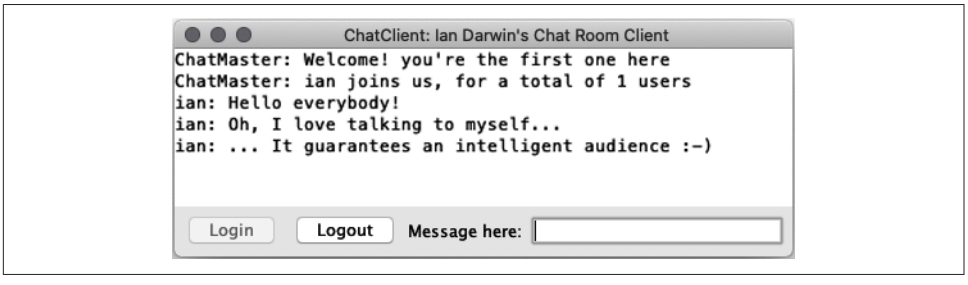

ChatClient: Ian Darwin's Chat Room Client
ChatMaster: Welcome! you're the first one here
ChatMaster: ian joins us, for a total of 1 users
ian: Hello everybody!
ian: Oh, I love talking to myself...
ian: ... It guarantees an intelligent audience :-)
Login
Logout
Message here:

Figure 14-2. Chat client in action

See Also

There are many better-structured ways to write a chat client, including WebSockets, XMPP, RMI, and JMS. XMPP is an open protocol for chat, with tools for Java and many other languages. RMI is Java's RPC interface and for many years was included in both Java SE and Java EE; it is not described in this edition of this book, but you can find the RMI chapter from previous editions on my website. The other technologies are part of Jakarta Enterprise, so you should refer to their documentation. If your communication goes over the public internet, you do need to encrypt your socket connection, so check out Sun's Java Secure Socket Extension (JSSE). If you took my earlier advice and used the standard HTTP protocol, you can encrypt the conversation just by changing the URL from `http` to `https`.

For a good overview of network programming from the C programmer's point of view, see the late W. Richard Stevens's *Unix Network Programming* (Prentice Hall). Despite the book's name, it's mainly about socket and TCP/IP/UDP programming and covers all parts of the (Unix) networking API and protocols such as TFTP in amazing detail.

CHAPTER 15

Server-Side Java

15.0 Introduction

Sockets form the underpinnings of almost all network software implementations. HTTP(S), JDBC, RMI, CORBA, EJB, XMPP, and the non-Java Remote Procedure Call (RPC) and Network File System (NFS): all are implemented by connecting various types of sockets together. Socket connections can be implemented in most any language, not just Java: C (the original), C++, Perl, and Python are also popular, and many others are possible. A client or server written in any one of these languages can communicate with its opposite written in any of the other languages. Therefore, it's worth taking a quick look at how the `ServerSocket` behaves, even if you wind up utilizing one of the higher-level services.

The discussion looks first at the `ServerSocket` itself, then at writing data over a socket in various ways. Finally, I show a complete implementation of a usable network server written in Java: the chat server from the client in the previous chapter.

Most production work in server-side Java uses the Jakarta Enterprise Edition (Jakarta EE, formerly Java EE), transferred years ago from Oracle to the Eclipse Software Foundation and renamed to Jakarta, but still widely referred to by its previous name (and occasionally by its very old name, "J2EE," which was retired some *20 years ago*). Those that don't use EE directly use some of its core APIs buried in other frameworks, like Spring Boot, Quarkus, Helidon, and more. Jakarta EE provides scalability and support for building well-structured, multitiered distributed applications. EE provides the servlet framework; a servlet is a strategy object that can be installed into any standard EE web server, and is the basis of two web view technologies: the older JavaServer Pages (JSP) and the newer, component-based JavaServer Faces (JSF). Allocation and injection of components is provided by the Contexts and Dependency Injection framework (CDI). Finally, EE provides a number of other network-based services, including EJB3 remote access and Java Messaging Service (JMS). These are outside the scope of this book; they are covered in other books, such as Arun Gupta's *Java EE 7 Essentials* (O'Reilly). This chapter is primarily for those who need or want to build their own server from the ground up.

15.1 Opening a Server Socket for Business

Problem

You need to write a socket-based server.

Solution

Create a `ServerSocket` for the given port number.

Discussion

The `ServerSocket` represents the *backend* (service end) of a connection, the server that waits patiently for clients to come along and connect to it. You construct a `ServerSocket` with just the port number.[6] Because it doesn't need to connect to another host, it doesn't need a particular host's address as the client socket constructor does.

6 You may not be able to pick just any port number for your own service, of course. Certain well-known port numbers are reserved for specific services and listed in your *services* file, such as 22 for Secure Shell and 25 for SMTP. Also, on server-based operating systems, ports below 1024 are considered privileged ports and require root or administrator privilege to create. This was an early security mechanism; today, with zillions of single-user desktops connected to the internet, it provides little real security, but the restriction remains.

Assuming the `ServerSocket` constructor doesn't throw an exception, you're in business. Your next step is to await client activity, which you do by calling `accept()`. This call blocks until a client connects to your server; at that point, the `accept()` returns to you a `Socket` object (not a `ServerSocket`) that is connected in both directions to the `Socket` object on the client (or its equivalent, if written in another language). Example 15-1 shows the code for a socket-based server.

Example 15-1. main/src/main/java/network/Listen.java

```
public class Listen {
  /** The TCP port for the service. */
  public static final short PORT = 9999;

  public static void main(String[] argv) throws IOException {
    ServerSocket sock;
    Socket  clientSock;
    try {
      sock = new ServerSocket(PORT);
      System.out.println("Listening on " + PORT);
      while ((clientSock = sock.accept()) != null) {

        // Process it, usually on a separate thread
        // to avoid blocking further accept() calls.
        process(clientSock);
      }

    } catch (IOException e) {
      System.err.println(e);
    }
  }

  /** This would do something with one client. */
  static void process(Socket s) throws IOException {
    System.out.println("Processing request from client " +
      s.getInetAddress());
    // The conversation would be here.
    s.close();
  }
}
```

You would normally use the same socket for both reading and writing, as shown in the next few recipes.

You may want to listen only on a particular network interface. Though we tend to think of network addresses as computer addresses, the two are not the same. A network address is actually the address of a particular network card, or network interface connection, on a given computing device. A desktop computer, laptop, tablet, or mobile phone might have only a single external interface (in addition to localhost/127.0.0.1), hence a single network address. But a large server machine might have two

or more external interfaces, usually when it is connected to several networks. A network router is a box, either special purpose (e.g., a commercial internet router) or general purpose (e.g., a Unix/Linux host), that has interfaces on multiple networks *and* has both the capability and the administrative permission to forward packets from one network to another. A program running on such a server machine might want to provide services only to its inside network or its outside network. One way to accomplish this is by specifying the network interface to be listened on. Suppose you want to provide a different view of web pages for your intranet than you provide to outside customers. For security reasons, you probably wouldn't run both these services on the same machine. But if you wanted to, you could do this by providing the network interface addresses as arguments to the `ServerSocket` constructor.

However, to use this form of the constructor, you don't have the option of using a string for the network address's name, as you did with the client socket; you must convert it to an `InetAddress` object. You also have to provide a `backlog` argument, which is the number of connection requests that can queue up to be accepted before clients are told that your server is too busy. The complete setup is shown in Example 15-2.

Example 15-2. main/src/main/java/network/ListenInside.java

```
public class ListenInside {
  /** The TCP port for the service. */
  public static final short PORT = 9999;
  /** The name of the network interface. */
  public static final String INSIDE_HOST = "acmewidgets-inside";
  /** The number of clients allowed to queue */
  public static final int BACKLOG = 10;

  public static void main(String[] argv) throws IOException {
    ServerSocket sock;
    Socket  clientSock;
    try {
      sock = new ServerSocket(PORT, BACKLOG,
        InetAddress.getByName(INSIDE_HOST));
      while ((clientSock = sock.accept()) != null) {

        // Process it.
        process(clientSock);
      }

    } catch (IOException e) {
      System.err.println(e);
    }
  }

  /** Hold server's conversation with one client. */
  static void process(Socket s) throws IOException {
```

```
    System.out.println("Connected from  " + INSIDE_HOST +
      ": " + s.getInetAddress(  ));
    // The conversation would be here.
    s.close();
  }
}
```

`InetAddress.getByName()`, described in Recipe 14.3, looks up the given hostname in a system-dependent way, typically referring to a configuration file in the */etc* or *\windows\System32\drivers\etc* directory, or to some kind of resolver such as the Domain Name System. Consult a good book on system and networking administration if you need to modify this data.

15.2 Finding Network Interfaces

Problem

You wish to find out about the computer's networking arrangements.

Solution

Use the `NetworkInterface` class.

Discussion

Every computer on a network has one or more *network interfaces*. On typical desktop machines, a network interface represents a network card or network port or some software network interface, such as the loopback interface. Each interface has an operating system–defined name. On most versions of Unix, these devices have a two- or three-character device driver name plus a digit (starting from 0), for example, `eth0` or `en0` for the first Ethernet on systems that hide the details of the card manufacturer, or `de0` and `de1` for the first and second Digital Equipment[7] DC21x4x-based Ethernet card, `xl0` for a 3Com EtherLink XL, and so on. The loopback interface is almost invariably `lo0` on all Unix-like platforms.

So what? Most of the time this is of no consequence to you. If you have only one network connection, like a cable link to your ISP, you really don't care. Where this matters is on a server, where you might need to find the address for a given network, for example. The `NetworkInterface` class lets you find this information. It has static methods for listing the interfaces and other methods for finding the addresses associated with a given interface. The program in Example 15-3 shows some examples of

7 Digital Equipment was absorbed by Compaq, which was then absorbed by HP, but the name remains `de` because the engineers who name such things don't care for corporate mergers anyway.

using this class. Running it prints the names of all the local interfaces. It then prints the machine's localhost network address and its associated interface.

Example 15-3. main/src/main/java/network/NetworkInterfaceDemo.java

```
public class NetworkInterfaceDemo {
  public static void main(String[] a) throws IOException {
    Enumeration<NetworkInterface> list = NetworkInterface.getNetworkInterfaces();
    while (list.hasMoreElements()) {
      // Get one NetworkInterface
      NetworkInterface iface = list.nextElement();
      // Print its name
      System.out.print(iface.getDisplayName());
      System.out.print(": ");
      Enumeration<InetAddress> addrs = iface.getInetAddresses();
      // And its address(es)
      while (addrs.hasMoreElements()) {
        InetAddress addr = addrs.nextElement();
        System.out.print(addr);
        System.out.print(' ');
      }
      System.out.println();

    }
    // Try to get the Interface for a given local (this machine's) address
    InetAddress destAddr = InetAddress.getByName("localhost");
    try {
      NetworkInterface dest = NetworkInterface.getByInetAddress(destAddr);
      System.out.println("Address for " + destAddr + " is " + dest);
    } catch (SocketException ex) {
      System.err.println("Couldn't get address for " + destAddr);
    }
  }
}
```

15.3 Returning a Response (String or Binary)

Problem

You need to write a string, binary, or object data to the client.

Solution

The socket gives you an `InputStream` and an `OutputStream`. Use them.

Discussion

The client socket examples in the previous chapter called the `getInputStream()` and `getOutputStream()` methods. These examples do the same. The main difference is

that these examples get the socket from a `ServerSocket`'s `accept()` method. Another distinction is that, by definition, the server normally creates or modifies the data and sends it to the client. Example 15-4 is a simple `Echo` server, which the `Echo` client of Recipe 14.5 can connect to. This server handles one complete connection with a client, then goes back and does the `accept()` to wait for the next client.

Example 15-4. main/src/main/java/network/EchoServer.java

```
public class EchoServer {
  /** Our server-side rendezvous socket */
  protected ServerSocket sock;
  /** The port number to use by default */
  public final static int ECHOPORT = 7;
  /** Flag to control debugging */
  protected boolean debug = true;

  /** main: construct and run */
  public static void main(String[] args) {
    int p = ECHOPORT;
    if (args.length == 1) {
      try {
        p = Integer.parseInt(args[0]);
      } catch (NumberFormatException e) {
        System.err.println("Usage: EchoServer [port#]");
        System.exit(1);
      }
    }
    new EchoServer(p).handle();
  }

  /** Construct an EchoServer on the given port number */
  public EchoServer(int port) {
    try {
      sock = new ServerSocket(port);
    } catch (IOException e) {
      System.err.println("I/O error in setup");
      System.err.println(e);
      System.exit(1);
    }
  }

  /** This handles the connections */
  protected void handle() {
    Socket ios = null;
    while (true) {
      try {
        System.out.println("Waiting for client...");
        ios = sock.accept();
        System.err.println("Accepted from " +
          ios.getInetAddress().getHostName());
```

```
        try (BufferedReader is = new BufferedReader(
              new InputStreamReader(ios.getInputStream(), "8859_1"));
            PrintWriter os = new PrintWriter(
              new OutputStreamWriter(ios.getOutputStream(), "8859_1"),
              true);) {
          String echoLine;
          while ((echoLine = is.readLine()) != null) {
            System.err.println("Read " + echoLine);
            os.print(echoLine + "\r\n");
            os.flush();
            System.err.println("Wrote " + echoLine);
          }
          System.err.println("All done!");
        }
      } catch (IOException e) {
        System.err.println(e);
      }
    }
  }
}
```

To send a string across an arbitrary network connection, some authorities recommend sending both the carriage return and the newline character; many protocol specifications require that you do so. This explains the `\r\n` in the code. If the other end is a DOS program or a Telnet-like program, it may be expecting both characters. On the other hand, if you are writing both ends, you can simply use `println()`—followed always by an explicit `flush()` before you read—to prevent the deadlock of having both ends trying to read with one end's data still in the `PrintWriter`'s buffer!

If you need to process binary data, use the data streams from `java.io` instead of the readers/writers. A server for the `DaytimeBinary` program of Recipe 14.6 should look like the following:

```
C:\javasrc\network>java network.DaytimeBinary
Remote time is 3161316799
BASE_DIFF is 2208988800
Time diff == 952284799
Time on localhost is Sun Mar 08 19:33:19 GMT 2014

C:\javasrc\network>time/t
Current time is  7:33:23.84p

C:\javasrc\network>date/t
Current date is Sun 03-08-2014

C:\javasrc\network>
```

Well, it happens that I have such a program in my arsenal, so I present it in Example 15-5. Note that it directly uses certain public constants defined in the client

class. Normally these are defined in the server class and used by the client, but I wanted to present the client code first.

Example 15-5. main/src/main/java/network/DaytimeServer.java

```
public class DaytimeServer {
  /** Our server-side rendezvous socket */
  ServerSocket sock;
  /** The port number to use by default */
  public final static int PORT = 37;

  /** main: construct and run */
  public static void main(String[] argv) {
    new DaytimeServer(PORT).runService();
  }

  /** Construct a DaytimeServer on the given port number */
  public DaytimeServer(int port) {
    try {
      sock = new ServerSocket(port);
    } catch (IOException e) {
      System.err.println("I/O error in setup\n" + e);
      System.exit(1);
    }
  }

  /** This handles the connections */
  protected void runService() {
    Socket ios = null;
    DataOutputStream os = null;
    while (true) {
      try {
        System.out.println("Waiting for connection on port " + PORT);
        ios = sock.accept();
        System.err.println("Accepted from " +
          ios.getInetAddress().getHostName());
        os = new DataOutputStream(ios.getOutputStream());
        long time = System.currentTimeMillis();

        time /= 1000;  // Daytime Protocol is in seconds

        // Convert to Java time base.
        time += TimeClient.BASE_DIFF;

        // Write it, truncating cast to int since it is using
        // the Internet Daytime Protocol which uses 4 bytes.
        // This will fail in the year 2038, along with all
        // 32-bit timekeeping systems based from 1970.
        // Remember, you read about the Y2038 crisis here first!
        // (the above comment added to this program 2001-03-30).
        os.writeInt((int)time);
```

```
        os.close();
      } catch (IOException e) {
        System.err.println(e);
      }
    }
  }
}
```

The program in Example 14-8 in the previous chapter reads a `Date` object over an `ObjectInputStream`. Example 15-6, the `DaytimeObjectServer` (the other end of that process), is a program that constructs a `Date` object each time it's connected to and returns it to the client.

Example 15-6. main/src/main/java/network/DaytimeObjectServer.java

```
public class DaytimeObjectServer {
  /** The TCP port for the object time service. */
  public static final short TIME_PORT = 1951;

  public static void main(String[] argv) {
    ServerSocket sock;
    Socket  clientSock;
    try {
      sock = new ServerSocket(TIME_PORT);
      while ((clientSock = sock.accept()) != null) {
        System.out.println("Accept from " +
          clientSock.getInetAddress());
        ObjectOutputStream os = new ObjectOutputStream(
          clientSock.getOutputStream());

        // Construct and write the Object
        os.writeObject(LocalDateTime.now());

        os.close();
      }

    } catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

15.4 Handling Multiple Clients

Problem

Your server needs to handle multiple clients.

Solution

Use a thread for each.

Discussion

In the C world, several mechanisms allow a server to handle multiple clients. One is to use a special system call `select()` or `poll()`, which notifies the server when any of a set of file/socket descriptors is ready to read, ready to write, or has an error. By including its rendezvous socket (equivalent to our `ServerSocket`) in this list, the C-based server can read from any of a number of clients in any order. Java does not provide this call, because it is not readily implementable on some Java platforms. Instead, Java uses the general-purpose `Thread` mechanism, as described in Chapter 11 (threads are now commonplace in many programming languages, though not always under that name). Each time the code accepts a new connection from the `ServerSocket`, it immediately constructs and starts a new thread object to process that client.[8]

The Java code to implement accepting on a socket is pretty simple, apart from having to catch `IOExceptions`:

```
/** Run the main loop of the Server. */
void runServer( ) {
    while (true) {
        try {
            Socket clntSock = sock.accept( );
            new Handler(clntSock).handle( );
        } catch(IOException e) {
            System.err.println(e);
        }
    }
}
```

To use a thread, you should implement `Runnable`. The `Handler` class creates a `Runnable` containing the socket, and submits it to a thread pool (see Recipe 11.1):

```
threadPool.submit(new Handler(clntSock));
```

But as written, `Handler` is constructed using the normal socket returned by `accept()` and normally calls the socket's `getInputStream()` and `getOutputStream()` methods and holds its conversation in the usual way. I'll present a full implementation, a threaded `Echo` client. First, a session showing it in use:

8 There are some limits to how many regular threads you can have running; if you expect to have thousands of threads running, particularly in a networked application, consider using virtual threads (see Recipe 11.2).

```
$ java network.EchoServerThreaded
EchoServerThreaded ready for connections.
Socket starting: Socket[addr=localhost/127.0.0.1,port=2117,localport=7]
Socket starting: Socket[addr=darian/192.168.1.50,port=13386,localport=7]
Socket starting: Socket[addr=darian/192.168.1.50,port=22162,localport=7]
Socket ENDED: Socket[addr=darian/192.168.1.50,port=22162,localport=7]
Socket ENDED: Socket[addr=darian/192.168.1.50,port=13386,localport=7]
Socket ENDED: Socket[addr=localhost/127.0.0.1,port=2117,localport=7]
```

Here I connected to the server once with my `EchoClient` program and, while still connected, called it up again (and again) with an operating system–provided Telnet client. The server communicated with all the clients concurrently, sending the messages from the first client back to the first client, and the data from the second client back to the second client, and so on. In short, it works. I ended the sessions with the end-of-file character in the program and used the normal disconnect mechanism from the Telnet client. Example 15-7 is the code for the server.

Example 15-7. main/src/main/java/network/EchoServerThreaded.java

```
public class EchoServerThreaded {

  final static ExecutorService threadPool =
    Executors.newVirtualThreadPerTaskExecutor();
  public static final int ECHOPORT = 2007;

  public static void main(String[] av) {
    new EchoServerThreaded().runServer();
  }

  public void runServer() {
    ServerSocket sock;
    Socket clientSocket;

    try {
      sock = new ServerSocket(ECHOPORT);

      System.out.println("EchoServerThreaded ready for connections.");

      /* Wait for a connection */
      while (true) {
        clientSocket = sock.accept();
        /* Create a thread to do the communication, and start it */
        new Handler(clientSocket).handle();
      }
    } catch (IOException e) {
      /* Crash the server if IO fails. Something bad has happened */
      System.err.println("Could not accept " + e);
      System.exit(1);
    }
  }
```

```
  /** A Thread subclass to handle one client conversation. */
  class Handler {
    final Socket sock;

    Handler(Socket s) {
      sock = s;
    }

    public void handle() {
      System.out.println("Socket starting: " + sock);
      threadPool.submit( () -> {
        try (BufferedReader is = new BufferedReader(
              new InputStreamReader(sock.getInputStream()));
            PrintStream os = new PrintStream(
              sock.getOutputStream(), true);) {
          String line;
          while ((line = is.readLine()) != null) {
            os.print(line + "\r\n");
            os.flush();
          }
          sock.close();
        } catch (IOException e) {
          System.out.println("IO Error on socket " + e);
          return;
        }
        System.out.println("Socket ENDED: " + sock);
      });
    }
  }
}
```

This version uses the recently minted virtual threads mechanism to hopefully cope with a lot of clients. All we have to do to use these threads is swap in the `Executors.newVirtualThreadPerTaskExecutor()` call as the first statement in the class. Using traditional `Threads` would have involved significant overhead, with every connection creating a new threaded object and destroying it when done. If you need that for some reason, just use one of the other `static new…Executor()` methods in the `ExecutorService`.

The performance of the virtual threads pool approach is reasonably good. For a small number of concurrent users, you might use a thread pool using conventional threads. It is also possible to implement a server of this sort with NIO, the "new" (back in Java 1.4) I/O package. However, the code to do so is more involved than anything in this chapter, and it is fraught with issues. There are several good tutorials on the internet for the person who needs to explore the performance of using NIO to manage server connections.

15.5 Serving the HTTP Protocol

Problem

You want to serve up a protocol such as HTTP.

Solution

Create a `ServerSocket` and write some code that speaks the particular protocol. Or in Java 18+, use the standard `SimpleFileServer`. You could also use a Java-powered web server such as Apache Tomcat or an Enterprise Edition (Jakarta EE) server such as JBoss WildFly.

Discussion

You can implement your own HTTP protocol server for very simple applications, which we'll do here. For basic applications, on Java 18 and later, you can also use the JDK tool `jwebserver` or its main class `SimpleFileServer`. For any serious development, you want to use Jakarta EE; see the note at the beginning of this chapter.

This example just constructs a `ServerSocket` and listens on it. When connections come in, they are replied to using the HTTP protocol. So it is somewhat more involved than the simple `Echo` server presented in Recipe 15.3. However, it's not a complete web server; the filename in the request is ignored, and a standard message is always returned. This is thus a *very* simple web server; it follows only the bare minimum of the HTTP protocol needed to send its response back. For a real web server written in Java, get Tomcat from the Apache Tomcat website or use any of the Jakarta/Java EE application servers. The code shown in Example 15-8, however, is enough to understand how to build a simple server that responds to requests using a protocol.

Example 15-8. main/src/main/java/network/WebServer0.java

```
public class WebServer0 {
  public static final int HTTP_PORT = 8080;
  public static final String CRLF = "\r\n";
  ServerSocket serverSock;
  /** A link to the source of this program, used in error message */
  static final String VIEW_SOURCE_URL =
  "https://github.com/IanDarwin/javasrc/tree/master/main/src/main/java/network";

  /**
   * Main method, just creates a server and calls its runServer().
   */
  public static void main(String[] args) throws Exception {
    var port = args.length == 1 ?  Integer.parseInt(args[0]) : HTTP_PORT;
```

```
  WebServer0 w = new WebServer0(port);
  System.out.println(
    "DarwinSys Java WebServer0 listening on http://localhost:" + port + "/");
  w.runServer();    // never returns!!
}

WebServer0() throws IOException {
  this(HTTP_PORT);
}

WebServer0(int port) throws IOException {
  this(new ServerSocket(port));
}

WebServer0(ServerSocket serverSock) {
  this.serverSock = serverSock;
}

/** RunServer accepts connections and passes each one to a handler. */
public void runServer() throws Exception {
  while (true) {
    try {
      Socket us = serverSock.accept();
      handle(us);
    } catch(IOException e) {
      System.err.println(e);
      return;
    }

  }
}

/** handle() handles one conversation with a Web client.
 * This is the only part of the program that "knows" HTTP.
 */
public void handle(Socket s) {
  BufferedReader is;  // inputStream, from Viewer
  PrintWriter os;    // outputStream, to Viewer
  try {
    String from = s.getInetAddress().toString();
    System.out.println("Accepted connection from " + from);
    is = new BufferedReader(new InputStreamReader(s.getInputStream()));
    String request = is.readLine();
    System.out.printf("Request: %s from %s\n", request, from);

    os = new PrintWriter(s.getOutputStream(), true);
    os.print("HTTP/1.0 200 Here is your data" + CRLF);
    os.print("Content-type: text/html" + CRLF);
    os.print("Server-name: DarwinSys Non-functional Java WebServer0" + CRLF);
    final String replyTemplate = """
      <html>
      <head>
```

```
          <title>Not a real server.</title>
          <meta name='viewport' content='width=device-width, initial-scale=1'/>
        </head>
        <body>
        <h1>Welcome, %s, but...</h1>
        <p>You have reached a desktop machine
        that does not run a real Web server.
        <p>Please pick another system!</p>
        <p>Or view <a href='%s'>
        this WebServer0 source on github</a>
        (scroll down to WebServer0.java).</p>
        <hr/>
        <em>Java-based WebServer0</em><hr/>
        </body>
        </html>
        """;
      String reply = replyTemplate.formatted(from, VIEW_SOURCE_URL);
      os.print("Content-length: " + reply.length() + CRLF);
      os.print(CRLF);
      os.print(reply + CRLF);
      os.flush();
      s.close();
    } catch (IOException e) {
      System.out.println("IOException " + e);
    }
    return;
  }
}
```

18 Using the predefined web server is quite simple and handles the HTTP(S) protocol for you. To use it from the command line, use the `jwebserver` command. To use it programmatically, you need only specify the location where the files live. Example 15-9 contains the code to do this.

This API's classes, based around `SimpleServer`, are in the *com.sun* namespace, meaning it is unsupported and subject to change. Further, since Oracle donated the Servlet API to the Eclipse Foundation as part of what is now Jakarta, they seem to have had to reimplement large parts of it, albeit with the `Exchange` class replacing the `HttpServletRequest` and `HttpServletResponse` classes.

Example 15-9. main/src/main/java/netweb/SimpleServerDemo.java

```
  void main() {
    var addr = new InetSocketAddress(PORT);
    var path = Path.of(System.getProperty("user.home") + "/public_html");
    var server = SimpleFileServer.createFileServer(addr, path, OutputLevel.INFO);
    System.out.printf("Starting listening on port %d to serve %s\n", PORT, path);
```

```
    server.start();
  }
```

Running it looks simlar to running the hand-rolled version earlier in this recipe:

```
$ java netweb/SimpleServerDemo.java
Starting listening on port 8910 to serve /home/ian/public_html
127.0.0.1 - - [22/Jul/2024:14:08:24 -0400] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [22/Jul/2024:14:08:25 -0400] "GET /favicon.ico HTTP/1.1" 404 -
^C
```

The directory *~/public_html* was once commonly utilized—at the dawn of the web, before social media, almost everybody had their own web pages at that location on some server or another. A URL path beginning with *~ian* would then map to the *public_html* folder in my home directory. Figure 15-1 shows the output of the preceding code generated in a standard browser.

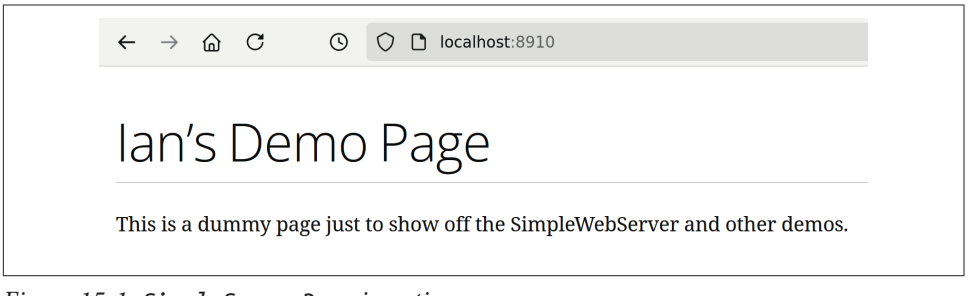


Figure 15-1. `SimpleServerDemo` in action

We can make it fancier by implementing the `Handler` portion of this server; this is shown in *main/src/main/java/netweb/SimpleServerDemo2.java*, which implements an `HttpHandler`, but doesn't have the code needed to map URLs to files. It instead looks at the request URL to check for bad-actor exploit attempts, and acts accordingly. The designers of the API would prefer that that kind of checking be done in their `Filter` class, but I wanted to keep the example short.

15.6 Securing a Web Server with TLS (formerly SSL) and JSSE

Problem

You want to protect your network traffic from prying eyes or malicious modification while the data is in transit.

Solution

Use the Java Secure Socket Extension (JSSE) to encrypt your traffic.

Discussion

JSSE provides services at a number of levels, but the easiest way to use it is simply to get your `ServerSocket` from an `SSLServerSocketFactory` instead of using the `ServerSocket` constructor directly. SSL is the Secure Sockets Layer; a revised version is known as TLS, Transport Layer Security. It is specifically for use on the web. To secure other protocols, you'd have to use a different form of the `SocketFactory`.

The `SSLServerSocketFactory` returns a `ServerSocket` that is set up to do SSL encryption. Example 15-10 uses this technique to override the `getServerSocket()` method in Recipe 15.5. If you're thinking this is too easy, you're wrong! The code is easy, but certificate management also requires work.

Example 15-10. main/src/main/java/network/JSSEWebServer0

```
/**
 * JSSEWebServer - subclass trivial WebServer0 to make it use SSL.
 * N.B. You MUST have set up a server certificate (see the
 * accompanying book text), or you will get the dreaded
 * javax.net.ssl.SSLHandshakeException: no cipher suites in common
 * (because without it JSSE can't use any of its built-in ciphers!).
 */
public class JSSEWebServer0 extends WebServer0 {

  public static final int HTTPS_PORT = 8443;

  public JSSEWebServer0() throws IOException {
    super();
  }

  public static void main(String[] args) throws Exception {
    if (System.getProperty("javax.net.ssl.keyStore") == null) {
      System.err.println(
        "You must pass in a keystore via -D; see the documentation!");
      System.exit(1);
    }
    System.out.println("DarwinSys JSSE Server 0.0 starting...");
    SSLServerSocketFactory ssf =
      (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
    var sock = ssf.createServerSocket(HTTPS_PORT);
    WebServer0 w = new WebServer0(sock);
    w.runServer();    // never returns!!
  }
}
```

That is, indeed, all the Java code one needs to write. You do have to set up an SSL certificate. For demonstration purposes, this can be a self-signed certificate; the steps are outlined on my website (steps 1–4 will suffice). You have to tell the JSSE layer where to find your keystore:

```
java -Djavax.net.ssl.keyStore=/home/ian/.keystore -Djavax.net.ssl.
keyStorePassword=secrit JSSEWebServer0
```

The typical client browser raises its eyebrows at a self-signed certificate (see Figure 15-2), but will accept the certificate if a user okays it. If you need a "real" certificate, check out Let's Encrypt, a nonprofit providing browser-accepted certificates at no cost.

Figure 15-3 shows the output of `WebServer0` being displayed over the HTTPS protocol (notice the padlock in the lower-right corner).

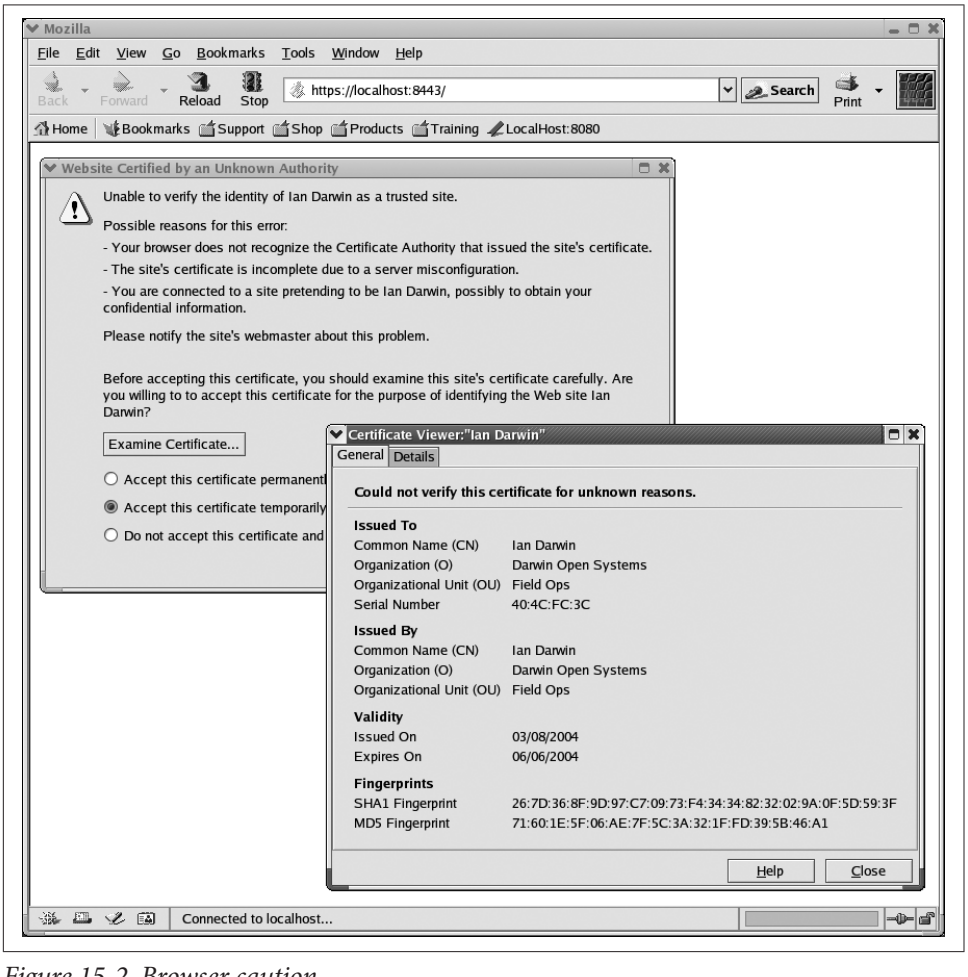

Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
https://localhost:8443/
Search
Print
Home Bookmarks Support Shop Products Training LocalHost:8080
Website Certified by an Unknown Authority
Unable to verify the identity of Ian Darwin as a trusted site.
Possible reasons for this error:
- Your browser does not recognize the Certificate Authority that issued the site's certificate.
- The site's certificate is incomplete due to a server misconfiguration.
- You are connected to a site pretending to be Ian Darwin, possibly to obtain your confidential information.
Please notify the site's webmaster about this problem.
Before accepting this certificate, you should examine this site's certificate carefully. Are you willing to to accept this certificate for the purpose of identifying the Web site Ian Darwin?
Examine Certificate...
Accept this certificate permanent
Accept this certificate temporarily
Do not accept this certificate and
Certificate Viewer:"Ian Darwin"
General Details
Could not verify this certificate for unknown reasons.
Issued To
Common Name (CN) Ian Darwin
Organization (O) Darwin Open Systems
Organizational Unit (OU) Field Ops
Serial Number 40:4C:FC:3C
Issued By
Common Name (CN) Ian Darwin
Organization (O) Darwin Open Systems
Organizational Unit (OU) Field Ops
Validity
Issued On 03/08/2004
Expires On 06/06/2004
Fingerprints
SHA1 Fingerprint 26:7D:36:8F:9D:97:C7:09:73:F4:34:34:82:32:02:9A:0F:5D:59:3F
MD5 Fingerprint 71:60:1E:5F:06:AE:7F:5C:3A:32:1F:FD:39:5B:46:A1
Help
Close
Connected to localhost...

Figure 15-2. Browser caution

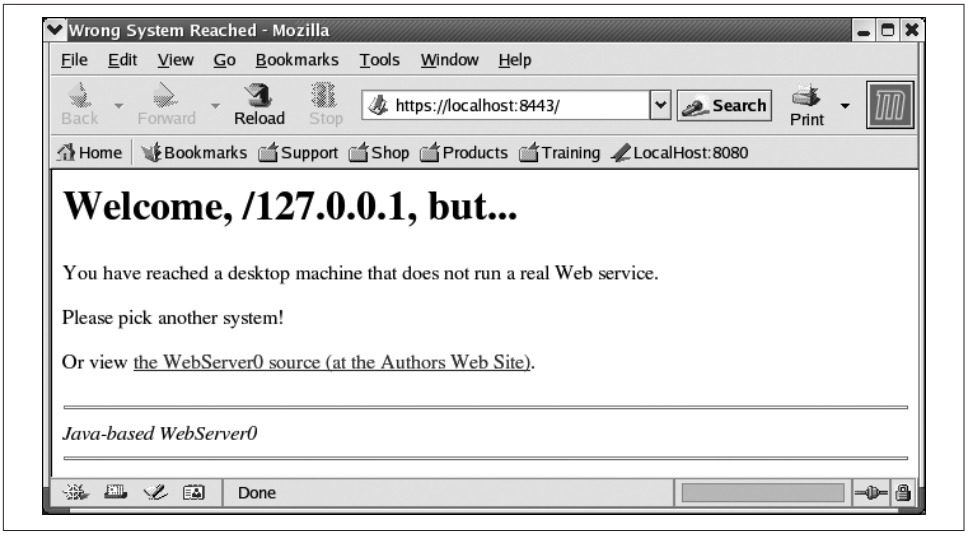

Wrong System Reached - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
https://localhost:8443/
Search
Print
Home Bookmarks Support Shop Products Training LocalHost:8080
Welcome, /127.0.0.1, but...
You have reached a desktop machine that does not run a real Web service.
Please pick another system!
Or view the WebServer0 source (at the Authors Web Site).
Java-based WebServer0
Done

Figure 15-3. With encryption

See Also

JSSE can do much more than encrypt web server traffic; this is, however, sometimes seen as its most exciting application. For more information on JSSE, see the Oracle reference or *Java Security* by Scott Oaks (O'Reilly).

15.7 Creating a REST Service/Microservice with JAX-RS

Problem

You want to implement a RESTful server by using the provided Java EE/Jakarta EE APIs.

Solution

Use JAX-RS annotations on a class that provides a service; install it in an enterprise application server. Or use a framework that provides its own micro HTTP server.

Discussion

This operation consists of both coding and configuration. The latter can be done by some of the higher-level frameworks listed at the end of this recipe.

The coding steps consist of creating a class that extends the JAX-RS `Application` class and adding annotations to a "resource" class that provides a service.

Here is a basic `Application` class:

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("")
public class RestApplication extends Application {
        // Empty
}
```

Example 15-11 is a "Hello, World"–type service class with the annotations needed to make it a service class and to have three sample methods.

Example 15-11. restdemo/src/main/java/rest/RestService.java

```
@Path("")
@ApplicationScoped
public class RestService {

  public RestService() {
    System.out.println("RestService.init()");
  }

  @GET @Path("/timestamp")
  @Produces(MediaType.TEXT_PLAIN)
  public String getDate() {
    return LocalDateTime.now().toString();
  }

  /** A Hello message method
   */
  @GET @Path("/greeting/{userName}")
  @Produces("text/html")
  public String doGreeting(@PathParam("userName")String userName) {
    System.out.println("RestService.greeting()");
    if (userName == null || userName.trim().length() <= 3) {
      return "Missing or too-short username";
    }
    return String.format("<h1>Welcome %s</h1><p>%s, We're glad to see you back!",
      userName, userName);
  }

  /** Used to download all items */
  @GET @Path("/names")
  @Produces(MediaType.APPLICATION_JSON)
  public List<String> findTasksForUser() {
    return List.of("Robin", "Jedunkat", "Lyn", "Glen");
  }
}
```

Now the class must be deployed. If we have created a proper Maven project structure (see Recipe 2.4) and have provided an application-server-specific Maven plug-in, and our development server is running, we can use some variation on `mvn deploy`. In the present example I have set this up, in the *rest* subdirectory, for deployment to WildFly, a Java Enterprise server from the JBoss open source community, funded by Red Hat, Inc. I need only type `mvn wildfly:deploy` to have the application compiled, packaged, and deployed to my server.

Once the service is deployed, and prior to writing a proper client, you can explore it interactively with a browser or, for simple GET requests, a Telnet client or `netcat`/`nc`:

```
$ telnet localhost 8080
Escape character is '^]'.
GET /rest/timestamp HTTP/1.0
Connection: keep-alive

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8

2025-10-16T19:54:31.42

GET /rest/greeting/Ian%20Darwin HTTP/1.0

HTTP/1.1 200 OK
Content-Type: text/html;charset=UTF-8

<h1>Welcome Ian Darwin</h1><p>Ian Darwin, We are glad to see you back!

get /rest/names HTTP/1.0
Accept: Application/JSON

HTTP/1.1 200 OK
Content-Type: application/json

["Robin","Jedunkat","Lyn","Glen"]
^] (CTRL/C)
$
```

An issue with REST is that there is not an official standard for documenting the API or protocol offered by a server (there are several competing specifications). People writing clients must either rely on plain documentation offered by the server's developers, or use trial and error to discover the protocol. Our example here is simple enough that we don't have this problem, but imagine a class with 20 or 30 methods in it.

REST frameworks

The two main midlevel toolkits that handle REST services (implementing JAX-RS) are RESTEasy and Jersey. Both are pretty much indistinguishable and are used by

many of the higher-level REST frameworks. Another specification is Eclipse MicroProfile, a subset ("profile") of the Jakarta EE spec intended for lightweight REST services/microservices.

The term *microservice architecture* (MSA) has become popular. A microservice is a small, self-contained service that "owns its own data," that is, can perform operations without having to wait for responses from a separate database server. (It's assumed that mutable data will eventually migrate to a central database, but this will be done in the background.) There are numerous articles and books on MSA; the architectural concerns are a bit out of scope for this book. Take a look at the O'Reilly book *Head First Software Architecture* by Raju Gandhi, Mark Richards, and Neal Ford.

The higher-level REST frameworks listed in Table 15-1 are free to use and reasonably popular. Each provides a complete framework for building and deploying REST services or microservices. That is, they both include the REST toolkit and provide a small, fast HTTP protocol server.

Table 15-1. REST/Microservice frameworks

Name	Home URL	Notes
Helidon	*https://helidon.io/*	Sponsored by Oracle
Quarkus	*https://quarkus.io*	Community framework
Micronaut	*https://micronaut.io/*	Community framework
Spring Boot	*https://spring.io/projects/spring-boot*	Plays well with other Spring packages

See Also

See this longer discussion on REST and the major frameworks (from 2020).

And if you're worried about the performance of Java versus other "cool new" languages, don't be. See this blog post from Mark Nelson that compares Java microservices to the Go language.

15.8 Unix Domain Sockets—Even on Windows! 16

Problem

You want to communicate more efficiently, and both the client and server are on the same machine.

Solution

Use the Unix domain socket mechanism.

Discussion

Unix domain sockets are an optimization for same-system networking. They were introduced into BSD Unix around 1979, but only appeared in Microsoft Windows in the 2010s. At the C programming level they work largely like network sockets. In Java they are best accessed with NIO buffers, whereas regular sockets can work directly with Strings. Otherwise their behavior is similar. They are especially useful in communication between an application server and its database; these often run on the same machine or VM.

This recipe shows both the client and the server because they share some code. The program is a simple server example, which sends a fixed string from the client to the server. Example 15-12 is the server portion.

Example 15-12. main/src/main/java/network/UnixDomainSocketServer.java

```
System.out.println("Unix Domain Sockets Demo");
Path socketPath = Path.of("/tmp/.jcb.socket");

System.out.println("Server side");
ServerSocketChannel serverChan =                    ❶
  ServerSocketChannel.open(StandardProtocolFamily.UNIX);
UnixDomainSocketAddress sockAddr =                  ❷
  UnixDomainSocketAddress.of(socketPath);
serverChan.bind(sockAddr);                          ❸
System.out.println("Waiting...");
SocketChannel channel;
while ((channel = serverChan.accept()) != null) {   ❹
  ByteBuffer inBuf = ByteBuffer.allocate(BUFSIZE);  ❺
  int numBytes;
  while ((numBytes = channel.read(inBuf)) > 0) {    ❻
    byte[] bytes = new byte[numBytes];
    inBuf.flip();                                   ❼
    inBuf.get(bytes);
    String message = new String(bytes);             ❽
    System.out.printf("[Incoming] %s\n", message);
    inBuf.clear();
  }
}
```

❶ Create the `ServerSocket` to accept connections from clients.

❷ Set up the IP address that clients will connect to.

❸ Hook up the IP address to the `ServerSocket`.

❹ Accept a connection from the client.

❺ Set up a buffer (chunk of memory) to read data into.

❻ Do the actual read, return number of bytes actually read, and quit the loop if we get no data (implies normal end-of-file condition).

❼ Create our own buffer to hold the data, and move the NIO buffer to a state where we can fetch from it.

❽ Create a `String` containing the bytes we read, and print it.

The corresponding client sender is shown in Example 15-13.

Example 15-13. main/src/main/java/network/UnixDomainSocketClient.java

```
System.out.println("Unix Domain Sockets Demo");
Path socketPath = Path.of("/tmp/.jcb.socket");

System.out.println("Client Starting");
UnixDomainSocketAddress sockAddr = UnixDomainSocketAddress.of(socketPath);
String message = "Hello via a UNIX Domain Socket";
try (SocketChannel channel = SocketChannel.open(StandardProtocolFamily.UNIX)) {
  channel.connect(sockAddr);
  ByteBuffer outBuf = ByteBuffer.allocate(BUFSIZE);
  outBuf.clear();
  outBuf.put(message.getBytes());
  outBuf.flip();
  System.out.printf("[Sending] %s\n", message);
  while (outBuf.hasRemaining()) {
    channel.write(outBuf);
  }
} catch (Exception ex) {
  throw new RuntimeException("Send failed: " + ex, ex);
}
```

When we run both programs:

```
$ java UnixDomainSocketServer.java &
[1] 42962
Unix Domain Sockets Demo
Server side
Waiting...
$ java UnixDomainSocketClient.java
Unix Domain Sockets Demo
Client Starting
[Sending] Hello via a UNIX Domain Socket
[Incoming] Hello via a UNIX Domain Socket
$
```

See Also

A good general reference on this chapter's topic is *Java Network Programming* by Elliotte Rusty Harold (O'Reilly).

The server side of any network mechanism is extremely sensitive to security issues. It is easy for one misconfigured or poorly written server program to compromise the security of an entire network! Of the many books on network security, two stand out: *Firewalls and Internet Security* by William R. Cheswick et al. (Addison-Wesley) and a series of books with *Hacking Exposed* in the title; the first in the series is by Stuart McClure et al. (McGraw-Hill).

This completes my discussion of server-side Java using sockets. A chat server could be implemented using several technologies, such as Remote Methods Invocation (RMI), an HTTP web service, Java Message Service (JMS), and a Java Enterprise API that handles store-and-forward message processing. This is beyond the scope of this book, but there's an example of an RMI chat server in the *chat* folder of the source distribution, and there's an example of a JMS chat server in *Java Message Service* by Mark Richards et al. (O'Reilly).

CHAPTER 16

Processing JSON Data

16.0 Introduction

JSON, or JavaScript Object Notation, is all of the following:

- A simple, lightweight data interchange format
- A simpler, lighter alternative to XML
- Easy to generate with `println()` or with one of several APIs
- Recognized directly by the JavaScript parser in all browsers, server-side node implementations, etc.
- Supported with add-on frameworks for all common languages (Java, C/C++, Perl, Ruby, Python, Lua, Erlang, Haskell, to name a few); a ridiculously long list of supported languages (including two dozen parsers for Java alone) is on the JSON home page

A basic JSON message might look like Example 16-1.

Example 16-1. json/src/main/resources/json/softwareinfo.json

```
{
  "name": "robinparse",
  "version": "1.2.3",
  "description": "Another Parser for JSON",
  "className": "RobinParse",
  "contributors": [
        "Robin Smythe",
        "Jon Jenz",
        "Jan Ardann"
    ]
}
```

As you can see, the syntax is simple, nestable, and amenable to human inspection.

The JSON home page provides a concise summary of JSON syntax. There are two kinds of structure: JSON objects (maps) and JSON arrays (lists). JSON *objects* are sets of name and value pairs, which can be represented in Java either as a `java.util.Map` *or* as the properties of a Java object. For example, the fields of a `LocalDate` (see Recipe 6.1) object for April 1, 2025, might be represented like this:

```
{
        "year": 2025,
        "month": 4,
        "day" : 1
}
```

JSON *arrays* are ordered lists, represented in Java either as arrays or as `java.util.Lists`. A list of two dates might look like this:

```
{
        [{
                "year": 2025,
                "month": 4,
                "day" : 1
        },{
                "year": 2025,
                "month": 5,
                "day" : 15
        }]
}
```

JSON is not whitespace-sensitive, so the preceding could also be written, with some loss of human readability but no loss of information or functionality, as this:

```
{[{"year":2025,"month":4,"day":1},{"year":2025,"month":5,"day":15}]}
```

Hundreds of parsers have, I'm sure, been written for JSON; a long list is at the middle of the JSON.org website. A few that come to mind for Java include the following:

`stringtree.org`
: Very small and lightweight

`org.json parser`
: Widely used because it's free and has a good domain name

`jackson.org parser`
: Widely used because it's very powerful and used with Spring Framework and with JBoss RESTEasy and WildFly

`jakarta.json` *a.k.a jsonp*
: The "official" Java standard

This chapter shows several ways of processing JSON data using some of the various APIs just listed. The official `jakarta.json` API is part of the Jakarta Enterprise API, so it is primarily used on the server side. This API uses some names that are also used by the `org.json` API, but not enough to be considered compatible; you can't easily use both in the same application.

16.1 Generating JSON Directly

Problem

You want to generate JSON without bothering to use an API.

Solution

Get the data you want, and use `println()` or `String.format()` as appropriate.

Discussion

If you are careful, you can generate JSON data yourself. For the utterly trivial cases, you can just use `PrintWriter.println()` or `String.format()`. For significant volumes, however, it's usually better to use one of the APIs.

This code prints the year, month, and date from a `LocalTime` object (see Recipe 6.1). Some of the JSON formatting is delegated to the `toJson()` method:

```
/**
 * Convert an object to JSON, not using any JSON API.
 * BAD IDEA - should use an API!
 */
public class LocalDateToJsonManually {

  private static final String OPEN = "{";
  private static final String CLOSE = "}";

  public static void main(String[] args) {
    LocalDate dNow = LocalDate.now();
    System.out.println(toJson(dNow));
  }

  public static String toJson(LocalDate dNow) {
    StringBuilder sb = new StringBuilder();
    sb.append(OPEN).append("\n");
    sb.append(jsonize("year", dNow.getYear()));
    sb.append(jsonize("month", dNow.getMonth()));
    sb.append(jsonize("day", dNow.getDayOfMonth()));
    sb.append(CLOSE).append("\n");
    return sb.toString();
  }
```

```
    public static String jsonize(String key, Object value) {
      return String.format("\"%s\": \"%s\",\n", key, value);
    }
  }
```

Of course, this is an extremely basic example. For anything more involved, or for the common scenario of having to parse JSON objects, using one of the frameworks will be easier on your nerves. That said, for some applications, the DIY approach may be faster. Time it and see!

16.2 Parsing and Writing JSON with Jackson

Problem

You want to read and/or write JSON using a full-function JSON API.

Solution

Use Jackson, the full-blown JSON API.

Discussion

Jackson provides many ways to work with JSON. For simple cases, you can have POJO (Plain Old Java Objects) converted to/from JSON more or less automatically, as is illustrated in Example 16-2.

You need the following three dependencies for full use of Jackson; later versions may be available when you read this:

```
com.fasterxml.jackson.core:jackson-core:2.13.0
com.fasterxml.jackson.core:jackson-annotations:2.13.0
com.fasterxml.jackson.core:jackson-databind:2.13.4.2
```

Example 16-2. json/src/main/java/json/ReadWriteJackson.java (reading and writing POJOs with Jackson)

```
import java.io.IOException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class ReadWriteJackson {


    public static void main(String[] args) throws IOException {
        ObjectMapper mapper = new ObjectMapper();                    ❶

        String jsonInput =                                           ❷
                "{\"id\":0,\"firstName\":\"Robin\",\"lastName\":\"Wilson\"}";
        Person q = mapper.readValue(jsonInput, Person.class);
```

```
        System.out.println("Read and parsed Person from JSON: " + q);

        Person p = new Person(0, "Roger", "Rabbit");              ❸
        System.out.print("Person object " + p +" as JSON = ");
        mapper.writeValue(System.out, p);
    }

    record Person(int id, String firstName, String lastName){    ❹
    @Override
    public String toString() {
      StringBuilder sb = new StringBuilder();
      if (firstName != null)
        sb.append(firstName).append(' ');
      if (lastName != null)
        sb.append(lastName);
      if (sb.length() == 0)
        sb.append("NO NAME");
      return sb.toString();
    }
  }
}
```

❶ Create a Jackson `ObjectMapper` that can map POJOs to/from JSON.

❷ Map the string `jsonInput` into a `Person` object with one call to `readValue()`.

❸ Convert the `Person` object `p` into JSON with one call to `writeValue()`.

❹ This class has its own `Person` type definition, a `record` with only three fields and a useful `toString()` method.

Running this example produces the following output:

```
Read and parsed Person from JSON: Robin Wilson
Person object Roger Rabbit as JSON = {"id":0,"firstName":"Roger",
        "lastName":"Rabbit","name":"Roger Rabbit"}
```

As another example, this code reads the example file that opened this chapter (which happens to have been a description of a JSON parser):

```
public class SoftwareParseJackson {
  final static String FILE_NAME = "/json/softwareinfo.json";

  public static void main(String[] args) throws Exception {
    ObjectMapper mapper = new ObjectMapper();                    ❶

    InputStream jsonInput =
      SoftwareParseJackson.class.getResourceAsStream(FILE_NAME);  ❷
    if (jsonInput == null) {
      throw new NullPointerException("can't find " + FILE_NAME);
    }
```

```
        SoftwareInfo sware = mapper.readValue(jsonInput, SoftwareInfo.class); ❸
        System.out.println(sware);
    }

}
```

❶ The `ObjectMapper` does the actual parsing of the JSON input.

❷ Reading the file using `getResourceAsStream()` (see Recipe 10.16).

❸ Converting the JSON string into a `domain.SoftwareInfo` object.

`SoftwareInfo` is a simple `record` type; notice the declaration `List<String>` for the contributors, which Jackson is quite capable of handling:

```
public record SoftwareInfo(String name,
  String version, String description,
  String className, List<String> contributors) {

  @Override
  public String toString() {
    return String.format("Software: %s (%s) by %s", name, version, contribu
tors);
  }
}
```

Running this example produces the following output:

```
Software: robinparse (1.2.3) by [Robin Smythe, Jon Jenz, Jan Ardann]
```

Of course there are situations where the mapping gets more involved; for this purpose, Jackson provides a set of annotations to control the mapping. But the default mapping is pretty good!

There is also a streaming API for Jackson, and even a large array of data format converters for formats other than JSON. Refer to the Jackson website for details.

16.3 Parsing and Writing JSON with org.json

Problem

You want to read/write JSON using a midsized, widely used JSON API.

Solution

Consider using the org.json API , also known as JSON-Java; it's widely used and is also used in Android.

Discussion

The org.json package is not as advanced as Jackson, nor as high level; it makes you think and work in terms of the underlying JSON abstractions instead of at the Java code level. For example, here is the org.json version of reading the software description from the opening of this chapter:

```
public class SoftwareParseOrgJson {
    final static String FILE_NAME = "/json/softwareinfo.json";

    public static void main(String[] args) throws Exception {

        InputStream jsonInput =
            SoftwareParseOrgJson.class.getResourceAsStream(FILE_NAME);
        if (jsonInput == null) {
            throw new NullPointerException("can't find" + FILE_NAME);
        }
        JSONObject obj = new JSONObject(new JSONTokener(jsonInput));      ❶
        System.out.println("Software Name: " + obj.getString("name"));    ❷
        System.out.println("Version: " + obj.getString("version"));
        System.out.println("Description: " + obj.getString("description"));
        System.out.println("Class: " + obj.getString("className"));
        JSONArray contribs = obj.getJSONArray("contributors");            ❸
        for (int i = 0; i < contribs.length(); i++) {                     ❹
            System.out.println("Contributor Name: " + contribs.get(i));
        }
    }

}
```

❶ Create the `JSONObject` from the input.

❷ Retrieve individual `String` fields.

❸ Retrieve the `JSONArray` of contributor names.

❹ `org.json.JSONArray` doesn't implement `Iterable`, so you can't use a `forEach` loop.

Running it produces the expected output:

```
Software Name: robinparse
Version: 1.2.3
Description: Another Parser for JSON
Class: RobinParse
Contributor Name: Robin Smythe
Contributor Name: Jon Jenz
Contributor Name: Jan Ardann
```

`JSONObject` and `JSONArray` use their `toString()` method to produce (correctly formatted) JSON strings, like this:

```
public class WriteOrgJson {
  public static void main(String[] args) {
    JSONObject jsonObject = new JSONObject();
    jsonObject.put("Name", "robinParse").     ❶
      put("Version", "1.2.3").
      put("Class", "RobinParse");
    String printable = jsonObject.toString(); ❷
    System.out.println(printable);
  }
}
```

❶ Nice that it offers a fluent API to allow chaining of method calls.

❷ `toString()` converts to textual JSON representation.

Running this produces the following:

```
{"Name":"robinParse","Class":"RobinParse","Version":"1.2.3"}
```

See Also

The org.json library code, including its Javadoc documentation, is available online at the JSON-java GitHub repository (under the name JSON-Java to differentiate it from the other packages).

16.4 Parsing and Writing JSON with JSON-B

Problem

You want to read/write JSON using a midsized, standards-conforming JSON API.

Solution

Consider using JSON-B, the new Java standard (JSR-367).

Discussion

The JSON-B (JSON Binding) API is designed to make it easy to read/write Java POJOs. This is neatly illustrated by the code in Example 16-3.

Example 16-3. json/src/main/java/json/ReadWriteJsonB.java (reading/writing JSON with JSON-B)

```
public class ReadWriteJsonB {

   public static void main(String[] args) throws IOException {

    Jsonb jsonb = JsonbBuilder.create();  ❶

    // Read
    String jsonInput =                    ❷
      "{\"id\":0,\"firstName\":\"Robin\",\"lastName\":\"Williams\"}";
    Person rw = jsonb.fromJson(jsonInput, Person.class);
    System.out.println(rw);

    String result = jsonb.toJson(rw);     ❸
    System.out.println(result);
   }
}
```

❶ Create a `Jsonb` object, your gateway to JSON-B services.

❷ Obtain a JSON string, and convert it to a Java object using `jsonb.fromJson()`.

❸ Convert a `Person` object back to a JSON string using the inverse `jsonb.toJson()`.

Note that the methods are sensibly named and that no annotations are needed on the Java entity class to make this work. However, there is an API that allows us to customize it. For example, the `fullName` property is really just a convenience for concatenating the first name and last name with a space in between. As such, it's completely redundant and does not need to be transmitted over a JSON network stream. However, running the program produces this output:

```
{"firstName":"Robin","fullName":"Robin Williams","id":0,"lastName":"Williams"}
```

We need only add the `@JsonbTransient` annotation to the `getFullName()` accessor in the `Person` class to eliminate the redundancy; running the program now produces this smaller output:

```
{"firstName":"Robin","id":0,"lastName":"Williams"}
```

See Also

As with most other JSON APIs, there is full support for customization, ranging from the simple annotation shown here up to writing complete custom serializer/deserializer helpers. See the JSON-B spec page and this longer tutorial online.

16.5 Finding JSON Elements with JSON Pointer

Problem

You have a JSON document and want to extract only selected values from it.

Solution

Use javax.json's implementation of JSON Pointer, the standard API for extracting selected elements from JSON.

Discussion

The Internet Standard RFC 6901 spells out in detail the syntax for JSON Pointer, a language-independent syntax for matching elements in JSON documents. Inspired by the XML syntax XPath, JSON Pointer is a bit simpler than XPath because of JSON's inherent simplicity. Basically, a JSON Pointer is a string that identifies an element (either simple or array) within a JSON document. The javax.json package provides an object model API somewhat similar to the XML DOM API for Java, letting you create immutable objects to represent objects (via `JsonObjectBuilder` and `JsonArrayBuilder`) or to read them from JSON string format via a `Reader` or `InputStream`.

JSON Pointers begin with a `/` (inherited from XPath), followed by the name of the element or subelement we want to look for. Suppose we extend our `Person` example from Example 16-3 to add an array of roles the comedian played, looking like this:

```
{"firstName":"Robin","lastName":"Williams",
        "age": 63,"id":0,
        "roles":["Mork", "Mrs. Doubtfire", "Patch Adams"]}
```

Then the following JSON Pointers should generate the given matches:

```
/firstName => Robin
/age => 63
/roles => ["Mork","Mrs. Doubtfire","Patch Adams"]
/roles/1 => "Mrs. Doubtfire"
```

The program in Example 16-4 demonstrates this. And its actual output is similar to what's expected:

```
/firstName => Robin
/age => 63; a org.eclipse.parsson.JsonNumberImpl$JsonIntNumber
/roles => ["Mork","Mrs. Doubtfire","Patch Adams"]
JsonArray roles.get(1) => "Mrs. Doubtfire"
/roles/1 => "Mrs. Doubtfire"
```

Example 16-4. json/src/main/java/json/JsonPointerDemo.java

```
public class JsonPointerDemo {

  public static void main(String[] args) {
    String jsonPerson = """
      {"firstName":"Robin","lastName":"Williams",
        "age": 63,
        "id":0,
        "roles":["Mork", "Mrs. Doubtfire", "Patch Adams"]
      }
    """;

    System.out.println("Input: " + jsonPerson);

    JsonReader rdr =
        Json.createReader(new StringReader(jsonPerson)); ❶
    JsonStructure jsonStr = rdr.read();
    rdr.close();

    JsonPointer jsonPointer = Json.createPointer("/firstName"); ❷
    JsonString jsonString = (JsonString)jsonPointer.getValue(jsonStr);
    String firstName = jsonString.getString();
    System.out.println("/firstName => " + firstName);

    JsonNumber num =                                          ❸
        (JsonNumber) Json.createPointer("/age").getValue(jsonStr);
    System.out.println("/age => " + num + "; a " + num.getClass().getName());

    jsonPointer = Json.createPointer("/roles");               ❹
    JsonArray roles = (JsonArray) jsonPointer.getValue(jsonStr);
    System.out.println("/roles => " + roles);
    System.out.println("JsonArray roles.get(1) => " + roles.get(1));

    jsonPointer = Json.createPointer("/roles/1");             ❺
    jsonString = (JsonString)jsonPointer.getValue(jsonStr);
    System.out.println("/roles/1 => " + jsonString);
  }
}
```

❶ Create the `JsonStructure`, the gateway into this API, from a `JsonReader`, using a `StringReader`.

❷ Create a JSON Pointer for the `firstName` element, and get the `JsonString` from the element's value. Since `getValue()` will throw an exception if the element is not found, use `jsonPointer.containsValue(jsonStr)` to check first, if you're not sure if the element will be found.

❸ Same for `age`, but using more fluent syntax. If you print the class name for the match in `/age`, it will report an implementation-specific implementation class, such as `org.glassfish.json.JsonNumberImpl$JsonIntNumber`. Change the age in the XML from 63 to 63.5 and it will print a class with `BigDecimal` in its name. Either way, calling `toString()` on this object will return just the numeric value.

❹ In the JSON file, `roles` is an array. Thus, getting it using a JSON Pointer should return a `JsonArray` object, so we cast it to a reference of that type. This behaves somewhat like an immutable `List` implementation, so we use the `get()` method. JSON array indices start at zero, as in Java.

❺ Retrieve the same array element directly, using a pattern with `/1` to mean the numbered element in the array.

It is possible (but fortunately not common) for a JSON element name to contain special characters such as a slash. Most characters are not special to JSON Pointer, but to match a name containing a slash (/), the slash must be entered as `~1`, and since that makes the tilde (~) special, tilde characters must be entered as `~0`. Thus if the Person JSON file had an element like `"ft/pt/~"`, you would look for it with `Json.createPointer("/ft~1pt~1~0");`.

See Also

The JSON Pointer API has additional methods that let you modify values and add/remove elements. The offical home page for javax.json, which includes JSON Pointer, is on the Jakarta EE website. That page also links to the Javadoc for javax.json.

Many APIs exist for Java. Jackson is the biggest and most powerful; org.json, `javax.json`, and JSON-B are in the middle; and StringTree (which I didn't give an example of because it doesn't have a Maven Artifact available) is the smallest. For a list of these and other JSON APIs, not just for Java, consult the JSON home page and scroll past the syntax summary.

CHAPTER 17

Reflection, or "A Class Named Class"

17.0 Introduction

The class `java.lang.Class` and the reflection package java.lang.reflect provide a number of mechanisms for gathering information from the Java Virtual Machine. Known collectively as *reflection*, these facilities allow you to load classes on the fly, to find methods and fields in classes, to generate listings of them, and to invoke methods on dynamically loaded classes. There is even a mechanism to let you construct a class from scratch (well, actually, from an array of bytes) while your program is running. This is about as close as Java lets you get to the magic, secret internals of the Java machine.

The JVM itself is a large program, normally written in C and/or C++, that implements the Java Virtual Machine abstraction. You can get the source for OpenJDK and other JVMs via the internet, and you could study it for months. Here we concentrate on just a few aspects, and only from the point of view of a programmer using the JVM's facilities, not how it works internally; that is an implementation detail that could vary from one vendor's JVM to another.

I'll start with loading an existing class dynamically, move on to listing the fields and methods of a class and invoking methods, and end by creating a class on the fly using a `ClassLoader`. One of the more interesting aspects of Java, and one that accounts for its flexibility (applets in days of yore, servlets, web services, and other dynamic APIs) while also once being part of its perceived speed problem, is the concept of *dynamic loading*. For example, even the simplest "Hello, Java" program has to load the class file for your `HelloJava` class, the class file for its parent (usually `java.lang.Object`), the class for `PrintStream` (because you used `System.out`), the class for `PrintStream`'s parent, and `IOException`, and its parent, and so on. To see this in action, try something like this:

```
java -verbose HelloJava | more
```

To give another example, when applets were popular, a browser would download an applet's bytecode file over the internet and run it on your desktop. How does it load the class file into the running JVM? We discuss this little bit of Java magic in Recipe 17.1. The chapter ends with replacement versions of the JDK tools `javap` and a cross-reference tool that you can use to become a famous Java author by publishing your very own reference to the complete Java API.

17.1 Loading and Instantiating a Class Dynamically

Problem

You want to load classes dynamically, just like web servers load your servlets.

Solution

Use `class.forName("ClassName");` and the class's `newInstance( )` method.

Discussion

Suppose you are writing a Java application and want other developers to be able to extend your application by writing Java classes that run in the context of your application. In other words, these developers are, in essence, using Java as an extension language. You would probably want to define a small set of methods that these extension programs would have and that you could call for such purposes as initialization, operation, and termination. The best way to do this is, of course, to publish a given, possibly abstract, class that provides those methods and get the developers to subclass from it.

We'll leave the thornier issues of security and of loading a class file over a network socket for now and assume that the user can install the classes into the application directory or into a directory that appears in the `CLASSPATH` at the time the program is run. First, let's define our class. We'll call it `Cooklet` (see Example 17-1) to avoid infringing on the overused word *applet*. Pretend each subclass will represent the code to drive some elaborate kind of food-preparing-and-cooking appliance through the steps of one traditional recipe. And we'll initially take the easiest path from ingredients to cookies before we complicate it.

Example 17-1. Cooklet.java

```
/** A simple class, just to provide the list of methods that
 * users need to provide to be usable in our application.
 * Note that the class is abstract so you must subclass it,
 * but the methods are nonabstract so you don't have to provide
```

```
 * dummy versions if you don't need a particular functionality.
 */
public abstract class Cooklet {

    /** The initialization method. The Cookie application will
     * call you here (AFTER calling your no-argument constructor)
     * to allow you to initialize your code.
     */
    public void initialize( ) {
    }

    /** The work method. The cookie application will call you
     * here when it is time for you to start cooking.
     */
    public void work( ) {
    }

    /** The termination method. The cookie application will call you
     * here when it is time for you to stop cooking and shut down
     * in an orderly fashion.
     */
    public void terminate( ) {
    }
}
```

Now, because we'll be baking, er, making this available to other people, we'll probably want to cook up a demonstration version too; see Example 17-2.

Example 17-2. main/src/main/java/reflection/DemoCooklet.java

```
public class DemoCooklet extends Cooklet {
  public void work() {
    System.out.println("I am busy baking cookies.");
  }
  public void terminate() {
    System.out.println("I am shutting down my ovens now.");
  }
}
```

But how does our application use it? Once we have the name of the user's class, we need to create a `Class` object for that class. This can be done easily using the static method `Class.forName()`. Then we can create an instance of it using the `Class` object's `newInstance()` method; this calls the class's no-argument constructor. Then we simply cast the newly constructed object to our `Cooklet` class, and we can call its methods! It actually takes longer to describe this code than to look at the code, so let's do that now; see Example 17-3.

Example 17-3. main/src/main/java/reflection/Cookies.java

```
public class Cookies {
  public static void main(String[] argv) {
    System.out.println("Cookies Application Version 0.0");
    Cooklet cooklet = null;
    String cookletClassName = argv[0];
    try {
      @SuppressWarnings("unchecked")
      Class<Cooklet> cookletClass =
        (Class<Cooklet>) Class.forName(cookletClassName);
      cooklet = cookletClass.getConstructor().newInstance();
    } catch (Exception e) {
      System.err.println("Error " + cookletClassName + e);
    }
    cooklet.initialize();
    cooklet.work();
    cooklet.terminate();
  }
}
```

And if we run it?

```
$ java Cookies DemoCooklet
Cookies Application Version 0.0
I am busy baking cookies.
I am shutting down my ovens now.
$
```

Of course, this version has rather limited error handling. But you already know how to fix that. Your `ClassLoader` can also place classes into a package by constructing a `Package` object; you should do this if loading any medium-sized set of application classes.

17.2 Printing Class Information

Problem

You want to print all the information about a class, similar to how it's done by `javap`.

Solution

Get a `Class` object, call its `getFields()` and `getMethods()`, and print the results.

Discussion

The JDK includes a program called `javap`, the Java printer. This normally prints the outline of a class file—a list of its methods and fields—but can also print out the Java bytecodes or machine instructions. The Kaffe package—a long gone

reimplementation of the JDK—did not include a version of `javap`, so I wrote one and contributed it (see Example 17-4). The Kaffe folks expanded it somewhat, but at the end it still worked basically the same. My version didn't print the bytecodes; it behaves rather like the original version when you don't give it any command-line options.

The `getFields()` and `getMethods()` methods return arrays of `Field` and `Method`, respectively; these are both in the java.lang.reflect package. I use a `Modifiers` object to get details on the permissions and storage attributes of the fields and methods. In many Java implementations, you can bypass this and simply call `toString()` in each `Field` and `Method` object (as I do here for `Constructors`). Doing it this way gives me a bit more control over the formatting.

Example 17-4. main/src/main/java/reflection/MyJavaP.java

```
public class MyJavaP {

  /** Simple main program, construct self, process each class name
   * found in argv.
   */
  public static void main(String[] argv) {
    MyJavaP pp = new MyJavaP();

    if (argv.length == 0) {
      System.err.println("Usage: MyJavaP className [...]");
      System.exit(1);
    } else for (int i=0; i<argv.length; i++)
      pp.doClass(argv[i]);
  }

  /** Format the fields and methods of one class, given its name.
   */
  protected void doClass(String className) {
    try {
      Class<? extends Object> c = Class.forName(className);

      final Annotation[] annotations = c.getAnnotations();
      for (Annotation a : annotations) {
        System.out.println(a);
      }

      System.out.println(c + " {");

      Field fields[] = c.getDeclaredFields();
      for (Field f : fields) {
        final Annotation[] fldAnnotations = f.getAnnotations();
        for (Annotation a : fldAnnotations) {
          System.out.println(a);
        }
```

```
        if (!Modifier.isPrivate(f.getModifiers()))
          System.out.println("\t" + f + ";");
      }

      Constructor<? extends Object>[] constructors = c.getConstructors();
      for (Constructor<? extends Object> con : constructors) {
        System.out.println("\t" + con + ";");
      }

      Method methods[] = c.getDeclaredMethods();
      for (Method m : methods) {
        final Annotation[] methodAnnotations = m.getAnnotations();
        for (Annotation a : methodAnnotations) {
          System.out.println(a);
        }
        if (!Modifier.isPrivate(m.getModifiers())) {
          System.out.println("\t" + m + ";");
        }
      }
      System.out.println("}");
    } catch (ClassNotFoundException e) {
      System.err.println("Error: Class " +
        className + " not found!");
    } catch (Exception e) {
      System.err.println("JavaP Error: " + e);
    }
  }
}
```

17.3 Getting a Class Descriptor

Problem

You want to get a `Class` object from a class name or instance, perhaps to display this information or to use it to access the class dynamically.

Solution

Use the compiler keyword `.class` or the `getClass()` method, as appropriate.

Discussion

If the type name is known at compile time, you can get the class instance using the compiler keyword `.class`, which works on any type that is known at compile time, even the eight primitive types.

Otherwise, if you have an object (an instance of a class), you can call the `java.lang.Object` method `getClass()`, which returns the `Class` object for the object's class (now that was a mouthful!):

```
        System.out.println("Trying the ClassName.class keyword:");
        System.out.println("Object class: " + Object.class);
        System.out.println("String class: " + String.class);
        System.out.println("String[] class: " + String[].class);
        System.out.println("LocalDate class: " + LocalDate.class);
        System.out.println("Current class: " + ClassKeyword.class);
        System.out.println("Class for int: " + int.class);
        System.out.println();

        System.out.println("Trying the instance.getClass() method:");
        System.out.println("Sir Robin the Brave".getClass());
        System.out.println(LocalDate.now().getClass());
```

When we run it, we see this:

```
C:\javasrc\reflect>java ClassKeyword
java main/src/main/java/reflection/ClassKeyword.java
Trying the ClassName.class keyword:
Object class: class java.lang.Object
String class: class java.lang.String
String[] class: class [Ljava.lang.String;
LocalDate class: class java.time.LocalDate
Current class: class reflection.ClassKeyword
Class for int: int

Trying the instance.getClass() method:
class java.lang.String
class java.time.LocalDate

C:\javasrc\reflect>
```

Nothing fancy, but as you can see, you can get the `Class` object for almost any type that is known at compile time, whether it's part of a package or not.

17.4 Finding and Using Methods and Fields

Problem

You need to find arbitrary method or field names in arbitrary classes.

Solution

Use the reflection package java.lang.reflect.

Discussion

If you just wanted to find fields and methods in one particular class, you wouldn't need this recipe; you could simply create an instance of the class using `new` and refer to its fields and methods directly. But the Reflection API allows you to find methods and fields in any class, even classes that have not yet been written! Given a class object

created as in Recipe 17.3, you can obtain a list of constructors, a list of methods, or a list of fields. The method `getMethods()` lists the methods available for a given class as an array of `Method` objects. Similarly, `getFields()` returns a list of `Field` objects. Because constructor methods are treated specially by Java, there is also a `getConstructors()` method, which returns an array of `Constructor` objects. Even though `Class` is in the package java.lang, the `Constructor`, `Method`, and `Field` objects it returns are in java.lang.reflect, so you need an import of this package. The `ListMethods` class (see Example 17-5) shows how to get a list of methods in a class when only its name is known at runtime.

Example 17-5. main/src/main/java/reflection/ListMethods.java

```
public class ListMethods {
  public static void main(String[] argv) throws ClassNotFoundException {
    if (argv.length == 0) {
      System.err.println("Usage: ListMethods className");
      return;
    }
    Class<?> c = Class.forName(argv[0]);
    Constructor<?>[] cons = c.getConstructors();
    printList("Constructors", cons);
    Method[] meths = c.getMethods();
    printList("Methods", meths);
  }
  static void printList(String s, Object[] o) {
    System.out.println("*** " + s + " ***");
    for (int i=0; i<o.length; i++)
      System.out.println(o[i].toString());
  }
}
```

For example, you could run Example 17-5 on a class like `java.lang.String` and get a fairly lengthy list of methods; I'll only show part of the output so you can see what it looks like:

```
> java reflection.ListMethods java.lang.String
*** Constructors ***
public java.lang.String( )
public java.lang.String(java.lang.String)
public java.lang.String(java.lang.StringBuffer)
public java.lang.String(byte[])
// and many more...
*** Methods ***
public static java.lang.String java.lang.String.copyValueOf(char[])
public static java.lang.String java.lang.String.copyValueOf(char[],int,int)
public static java.lang.String java.lang.String.valueOf(char)
// and more valueOf( ) forms...
public boolean java.lang.String.equals(java.lang.Object)
public final native java.lang.Class java.lang.Object.getClass( )
```

```
    // and more java.lang.Object methods...
    public char java.lang.String.charAt(int)
    public int java.lang.String.compareTo(java.lang.Object)
    public int java.lang.String.compareTo(java.lang.String)
```

You can see that this could be extended (almost literally) to write a `BeanMethods` class that would list only the set/get methods defined in a bean class.

Alternatively, you can find a particular method and invoke it, or find a particular field and refer to its value. Let's start by finding a given field, because that's the easiest. Example 17-6 is code that, given an `Object` and the name of a field, finds the field (gets a `Field` object) and then retrieves and prints the value of that `Field` as an `int`.

Example 17-6. main/src/main/java/reflection/FindField.java

```
public class FindField {

  public static void main(String[] unused)
  throws NoSuchFieldException, IllegalAccessException {

    // Create instance of FindField
    FindField gf = new FindField();

    // Create instance of target class (YearHolder defined below).
    Object o = new YearHolder();

    // Use gf to extract a field from o.
    System.out.println("The value of 'currentYear' is: " +
      gf.intFieldValue(o, "currentYear"));
  }

  int intFieldValue(Object o, String name)
  throws NoSuchFieldException, IllegalAccessException {
    Class<?> c = o.getClass();
    Field fld = c.getField(name);
    int value = fld.getInt(o);
    return value;
  }
}

/** This is just a class that we want to get a field from */
class YearHolder {
  /** Just a field that is used to show getting a field's value. */
  public int currentYear = LocalDate.now().getYear();
}
```

What if we need to find a method? The simplest way is to use the methods `getMethod()` and `invoke()`. But this is not altogether trivial. Suppose that somebody gives us a reference to an object. We don't know its class but have been told that it should have this method:

```
    public void work(String s) { }
```

We wish to invoke `work()`. To find the method, we must make an array of `Class` objects, one per item in the parameter list. So, in this case, we make an array containing only a reference to the class object for `String`. Because we know the name of the class at compile time, we'll use the shorter invocation `String.class` instead of `Class.forName()`. This, plus the name of the method as a string, gets us entry into the `getMethod()` method of the `Class` object.

If this succeeds, we have a `Method` object. But guess what? In order to invoke the method, we have to construct yet another array, this time an array of `Object` references actually containing the data to be passed to the invocation. We also, of course, need an instance of the class in whose context the method is to be run. For this demonstration class, we need to pass only a single string, because our array consists only of the string. Example 17-7 is the code that finds the method and invokes it.

Example 17-7. main/src/main/java/reflection/GetAndInvokeMethod.java

```
/**
 * Get a given method, and invoke it.
 */
public class GetAndInvokeMethod {

  /** This class is just here to give us something to work on,
   * with a println() call that will prove we got into it.
   */
  static class X {
    public void work(int i, String s) {
      System.out.printf("Called: i=%d, s=%s%n", i, s);
    }
    // The main code
    public static void main(String[] args) {
      System.out.println("Main.args = " + Arrays.toString(args));
    }
  }

  public static void main(String[] argv) {
    try {
      Class<?> clX = X.class; // or Class.forName("X");

      // To find a method we need the array of matching Class types.
      Class<?>[] argTypes = {
        int.class,
        String.class
      };

      // Now find a Method object for the given method.
      Method worker = clX.getMethod("work", argTypes);
```

```
        // To INVOKE the method, we need the invocation
        // arguments as an Object array.
        Object[] theData = {
          42,
          "Chocolate Chips"
        };

        // The obvious last step: invoke the method.
        // First arg is an instance, null if static method
        worker.invoke(new X(), theData);

        System.out.println("First Invoke Done");

        // Same deal but for main, a static method taking String[]:
        final Method m = clX.getMethod("main", String[].class);
        final Object[] args = new Object[1];
        args[0] = "Hello World Of Java".split(" ");
        // e.g., args[0] is itself an array
        m.invoke(null, args);
        System.out.println("Second Invoke Done");

      } catch (Exception e) {
        System.err.println("Invoke() failed: " + e);
        e.printStackTrace();
      }
    }
}
```

Not tiny, but it's still not bad. In most programming languages, you couldn't do that in the 40 lines it took us here.

When the arguments to a method are of a primitive type, such as `int`, you do not pass `Integer.class` into `getMethod()`. Instead, you must use the class object representing the primitive type `int`. The easiest way to find this class is in the `Integer` class, as a public constant named `TYPE`, so you'd pass `Integer.TYPE`. The same is true for all the primitive types; for each, the corresponding wrapper `class` has the primitive `class` referred to as `TYPE`.

Another caution: it's an advanced topic, but there are some interactions between the module system and the Reflection API related to permissions and intermodule reflection.

Java also includes a mechanism called a `MethodHandle` that was intended to both simplify and generalize the use of Reflection to invoke methods; we cover it in the following recipe, Recipe 17.5.

17.5 Invoking Class Members via MethodHandles

Problem

You hope to find an easy way to find and invoke methods, with the possibility of transforming arguments being passed into them.

Solution

Consider using the MethodHandle API. It is intended to replace the Reflection API for invoking methods, is a bit easier to use, and is generally faster.

Discussion

As with many such APIs, there is a lookup class that must be created initially. You then create a `MethodType` describing the signature (one return type and any number of argument types) of the method you are focused on. These are then used to look up a method and create a `MethodHandle` for it. You then call the methods using the `invoke()` or `invokeExact()` method on the `MethodHandle`, passing required arguments.

A simple example (after imports from `java.lang.invoke`) creates a `MethodType` for a method returning a string given two `int`s. On the next line we see that this is going to be used to invoke the `substring()` method, and finally we see the invocation, to extract the substring establishment from the longer string provided, as shown in Example 17-8.

Example 17-8. main/src/main/java/reflection/MethodHandleAbcs.java

```
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodType mt = MethodType.methodType(String.class, int.class, int.class);
MethodHandle mh = lookup.findVirtual(String.class, "substring", mt);
String s = (String) mh.invokeExact("Antidisestablishmentarianism", 7, 20);
System.out.println(s);  // prints "establishment"
```

Example 17-9 contains a more complete example.

Example 17-9. main/src/main/java/reflection/MethodHandleDemo.java

```
// Create a Lookup object
MethodHandles.Lookup lup = MethodHandles.lookup();

// First invoke LocalDate's static "of(year, month, day)" method:

// Create a matcher for the arg list types we want to invoke
// First the return type, then the argument types
```

```
MethodType mt = MethodType.methodType(LocalDate.class,
    int.class, int.class, int.class);
// Use the Lookup object to find the static method "of" in LocalDate
MethodHandle mh = lup.findStatic(LocalDate.class, "of", mt);
// At last, invoke the method!
LocalDate endDate = (LocalDate) mh.invokeExact(2014, 6, 10);
System.out.println("LocalDate from 'of' = " + endDate);

// Second, invoke the LocalDate's instance method toString()
// "toString()" returns a String, takes no args
mt = MethodType.methodType(String.class);
// Find the instance method in the LocalDate class
mh = lup.findVirtual(LocalDate.class, "toString", mt);
// Invoke it in the context of the LocalDate object created earlier
String asString = (String) mh.invokeExact(endDate);
System.out.println("LocalDate as String is " + asString);
```

As always, the Javadoc for the `MethodHandle` class contains more detail.

17.6 Listing Classes in a Package

Problem

You want to get a list of all the classes in a package.

Solution

You can't, in the general sense. There are some limited approaches, most involving `CLASSPATH` scanning.

Discussion

There is no way to find out all the classes in a package, in part because, as we just saw in Recipe 17.9, you can add classes to a package at any time! And, for better or worse, the JVM and standard classes such as `java.lang.Package` do not even allow you to enumerate the classes currently in a given package.

The nearest you can come is to look through the `CLASSPATH`. And this will surely work only for local directories and JAR files; if you have locally defined or network-loaded classes, this is not going to help. In other words, it will find compiled classes, but not dynamically loaded classes. There are several libraries that can automate this for you, and you're welcome to use them. The code to scan the `CLASSPATH` is fairly simple at heart, though, so classy developers with heart will want to examine it. Example 17-10 shows my `ClassesInPackage` class with its one static method. The code works but is rather short on error handling, and it will crash on nonexistent packages and other failures.

The code goes through a few gyrations to get the `CLASSPATH` as an enumeration of URLs, then looks at each element.

file
: URLs will contain the pathname of the file containing the *.class* file, so we can just list it.

jar
: URLs contain the filename as *file:/path_to_jar_file!package/name*, so we have to pull this apart; the *package/name* suffix is slightly redundant in this case because it's the package we asked the `ClassLoader` to give us.

Example 17-10. main/src/main/java/reflection/ClassesInPackage.java

```
public class ClassesInPackage {

  /** This approach began as a contribution by Paul Kuit at
   * http://stackoverflow.com/questions/1456930/, but his only
   * handled single files in a directory in classpath, not in Jar files.
   * N.B. Does NOT handle system classes!
   * @param packageName
   * @return
   * @throws IOException
   */
  public static String[] getPackageContent(String packageName)
    throws IOException {

    final String packageAsDirName = packageName.replace(".", "/");
    final List<String> list = new ArrayList<>();
    final Enumeration<URL> urls =
        Thread.currentThread().
        getContextClassLoader().
        getResources(packageAsDirName);
    while (urls.hasMoreElements()) {
      URL url = urls.nextElement();
      // System.out.println("URL = " + url);
      String file = url.getFile();
      switch (url.getProtocol()) {
      case "file":
        // This is the easy case: "file" is
        // the full path to the classpath directory
        File dir = new File(file);
        for (File f : dir.listFiles()) {
          list.add(packageAsDirName + "/" + f.getName());
        }
        break;
      case "jar":
        // This is the harder case; "file" is of the form
        // "jar:/home/ian/bleah/darwinsys.jar!com/darwinsys/io"
        // for some jar file that contains at least one class from
```

```
        // the given package.
        int colon = file.indexOf(':');
        int bang = file.indexOf('!');
        String jarFileName = file.substring(colon + 1, bang);
        JarFile jarFile = new JarFile(jarFileName);
        Enumeration<JarEntry> entries = jarFile.entries();
        while (entries.hasMoreElements()) {
          JarEntry e = entries.nextElement();
          String jarEntryName = e.getName();
          if (!jarEntryName.endsWith("/") &&
            jarEntryName.startsWith(packageAsDirName)) {
            list.add(jarEntryName);
          }
        }
        break;
      default:
        throw new IllegalStateException(
        "Dunno what to do with URL " + url);
      }
    }
    return list.toArray(new String[] {});
  }

  public static void main(String[] args) throws IOException {
    String[] names = getPackageContent("com.darwinsys.io");
    for (String name : names) {
      System.out.println(name);
    }
    System.out.println("Done");
  }
}
```

Note that if you run this application in the *javasrc* project, it will list the members of the demonstration package (com.darwinsys.io) twice, because it will find them both in the build directory and in the JAR file. If this is an issue, change the `List` to a `Set` (see Recipe 7.4)—the reason that different data structures exist!

17.7 Accessing Nested Members of Same Class

Problem

You want to access methods declared in your outer class or in another of its nested classes.

Solution

Access private members in containing classes just as you did before Java 11. Use the nest-based API to get information on nest mates.

Discussion

Java has long supported nested classes, also called inner classes. A problem arose when an inner class needed to access a private member of the outer class. Since this was forbidden by the JVM's access rules, the compiler had to insert a hidden "bridge" method in the outer class, to be called by the inner class. Example 17-11 shows an example of *nest mates*, a term that encompasses the host (outer) class and any nested (inner) classes.

Example 17-11. main/src/main/java/lang/NestMates.java

```
/**
 * Illustrate "nest mates" access control. Compile this on
 * both Java 1.8 and Java 11, and compare the .class files.
 */
public class NestMates {

  private void privateProcess() {
    System.out.println("Hello from a private method.");
  }

  public class Nested {
    public void innerProcess() {
      privateProcess();
    }
  }

  Nested nested = new Nested();

  public static void main(String[] args) {
    new NestMates().nested.innerProcess();
  }
}
```

The inner class invokes a private method in the outer class. Prior to Java 11, this was technically prohibited by the JVM specification. However, the Java compiler would insert a hidden "bridge" method in the outer class and generate a call to it whenever the inner class tried to call the outer private method. As of Java 11, the JVM rules were altered so that this access became legal—nest mates are allowed to touch each other's private members—and the bridge method is no longer needed.

The bottom line is that you don't have to change anything in this class when moving from Java 8 to Java 11—the compiler simply handles access differently.

You can view the change in the generated JVM bytecode by running Java 8 and a Java 11 or later `javac` command, and viewing the compiled code with `javap -c`. An example of this is in the shell script `nestmatesdemo` in the same directory as Example 17-11.

Since the "nest-mated-ness" may be of interest to some tools, the Reflection API was extended to allow explicit access to nest mates. The `Class` class added the following methods:

- `Class getNestHost()` returns the host of the nest.
- `boolean isNestMateOf(Class c)` returns `true` if `this` and `c` are in the same nest.
- `Class[] getNestMembers()` returns an array of `Class` for all members of the nest.

These are briefly demonstrated in Example 17-12.

Example 17-12. main/src/main/java/lang/NestMatesReflection.java

```
void main() {
  Class outer = lang.NestMates.class;
  Class inner = lang.NestMates.Nested.class;
  System.out.println("Is outer a nestmate of itself? " + outer.isNestmateOf(outer));
  System.out.println("Is outer a nestmate of inner? " + outer.isNestmateOf(inner));
  System.out.println("What is inner's nest host? " + inner.getNestHost());
  System.out.println("What are all of inner's nestmates? " +
    java.util.Arrays.toString(inner.getNestMembers()));
}
```

17.8 Accessing Private Methods and Fields via Reflection

Problem

You want to access private fields and have heard you can do so using the Reflection API.

Solution

It's generally a bad idea to access private fields. If you have to, you can. This mechanism is really intended for tool builders who make IDEs, debuggers, and certain libraries.

Discussion

There is occasionally a need to access private fields in other classes. For example, I did so recently while writing a JUnit test case that needed to see all the fields of a target class. The secret is to call the `Field` or `Method` descriptor's `setAccessible()` method, passing the value `true` before trying to get the value or invoke the method. It really is that easy, as shown in Example 17-13.

Example 17-13. main/src/main/java/reflection/DefeatPrivacy.java

```
class ClassWithPrivateField {
  @SuppressWarnings("unused") // Used surreptitiously below.
  private int p = 42;
  int q = 3;
}

/**
 * Demonstrate that it is, in fact, all too easy to access private members
 * of an object using Reflection.
 */
public class DefeatPrivacy {

  public static void main(String[] args) throws Exception {
    new DefeatPrivacy().process();
  }

  private void process() throws Exception {
    ClassWithPrivateField x = new ClassWithPrivateField();
    System.out.println(x);
    // System.out.println(x.p); // Won't compile
    System.out.println(x.q);
    Class<? extends ClassWithPrivateField> class1 = x.getClass();
    Field[] flds = class1.getDeclaredFields();
    for (Field f : flds) {
      f.setAccessible(true);  // bye-bye "private"
      System.out.println(f + "==" + f.get(x));
      f.setAccessible(false);  // reset to "correct" state
    }
  }
}
```

Use this with *extreme care*, because it can defeat some of the most cherished principles of Java programming.

17.9 Constructing a Class from Scratch with a ClassLoader

Problem

You need to load a class from a nonstandard location and run its methods.

Solution

Examine the existing loaders such as `java.net.URLClassLoader`. If none are suitable, write and use your own `ClassLoader`.

Discussion

A `ClassLoader`, of course, is a class that loads classes. A few `ClassLoaders` are built into the Java Virtual Machine, but your application can create others as needed. Learning to write and run a working `ClassLoader` and using it to load a class and run its methods is a nontrivial exercise. In fact, you rarely need to write a `ClassLoader`, but knowing how is helpful in understanding how the JVM finds classes, creates objects, and calls methods.

A given class loader will have a *namespace* associated with it. For example, in a Java web server, there can easily be multiple web applications, and several of them might have a class with the same full name. In a single basic application this would be a conflict, but with multiple `ClassLoaders`, one per web app, there is no conflict, as each has its own namespace.

`ClassLoader` itself is abstract; you must subclass it, presumably providing a `loadClass()` method that loads classes as you wish. It can load the bytes from a network connection, a local disk, RAM, a serial port, or anywhere else. Or you can construct the class file in memory yourself, if you have access to a compiler.

There is a general-purpose loader called `java.net.URLClassLoader` that can be used if all you need is to load classes via the web protocol (or, more generally, from one or more URLs).

You must call the `ClassLoader loadClass()` method for any classes you wish to explicitly load from it. Note that this method is called to load all classes required for classes you load (superclasses that aren't already loaded, for example). However, the JVM still loads classes that you instantiate with the `new` operator normally via `CLASSPATH`.

When writing a `ClassLoader`, your `loadClass()` method needs to get the class file into a byte array (typically by reading it), convert the array into a `Class` object, and return the result.

What? That sounds a bit like "and then a miracle occurs…" And it is. The miracle of class creation, however, happens down inside the JVM, where you don't have access to it. Instead, your `ClassLoader` has to call the `protected defineClass()` method in your superclass (which is `java.lang.ClassLoader`). This is illustrated in Figure 17-1, where a stream of bytes containing a hypothetical `Chicken` class is converted into a ready-to-run `Chicken` class in the JVM by calling the `defineClass()` method.

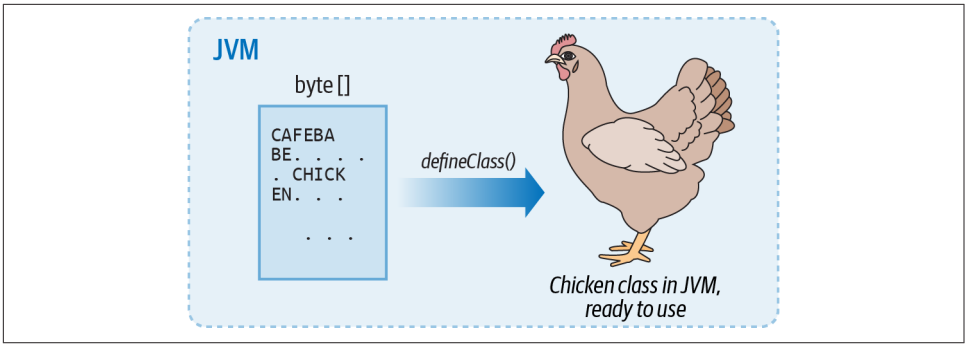


Figure 17-1. `ClassLoader` in action

What next?

To use your `ClassLoader` subclass, you need to instantiate it and call its `loadClass()` method with the name of the class you want to load. This gives you a `Class` object for the named class; the `Class` object in turn lets you construct instances, find and call methods, etc. Refer back to Recipe 17.4.

17.10 Constructing a Class from Scratch with JavaCompiler

Problem

You'd rather construct a class dynamically by generating and compiling source code.

Solution

Use the `JavaCompiler` from javax.tools.

Discussion

There are many situations in which you might need to generate code on the fly. If you're writing a framework, you might want to introspect on a model class to find its fields and generate accessors for them on the fly. As we've seen in Recipe 17.4, you can do this with the `Field` class. However, for a high-volume operation it may be more efficient to generate direct access code.

The Java Compiler API has been around since Java 1.6 and is fairly easy to use for simple cases. Here are the basic steps:

- Get the `JavaCompiler` object for your current Java runtime. If it's not available, either give up altogether or fall back to using reflection.

- Get a `CompilerTask` (which is also a `Callable`) to run the compilation, passing input and outputs.
- Invoke the `Callable`, either directly or by using an `ExecutorService`.
- Check the results. If true, invoke the class.

This is demonstrated in Example 17-14.

Example 17-14. main/src/main/java/reflection/JavaCompilerDemo.java

```
import javax.tools.JavaCompiler;
import javax.tools.SimpleJavaFileObject;
import javax.tools.ToolProvider;

/** Demo the Java Compiler API: Create a class, compile, load, and run it.
 * Best run standalone using "java JavaCompiler.java"
 */
public class JavaCompilerDemo {
  private final static String PACKAGE = "reflection";
  private final static String CLASS = "AnotherDemo";
  private static boolean verbose;
  public static void main(String[] args) throws Exception {
    String source = "package " + PACKAGE + ";\n" +                    ❶
      "public class " + CLASS + " {\n" +
      "\tpublic static void main(String[] args) {\n" +
      "\t\tString message = (args.length > 0 ? args[0] : \"Hi\")" + ";\n" +
      "\t\tSystem.out.println(message + \" from AnotherDemo\");\n" +
      "\t}\n}\n";
    if (verbose)
      System.out.print("Source to be compiled:\n" + source);

    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler(); ❷
    if (compiler == null) {
      throw new IllegalStateException("No default compiler, giving up.");
    }
    Callable<Boolean> compilation =
      compiler.getTask(null, null, null, List.of("-d","."), null, ❸
      List.of(new MySource(CLASS, source)));
    boolean result = compilation.call();                             ❹
    if (result) {
      System.out.println("Compiled OK");
      Class<?> c = Class.forName(PACKAGE + "." + CLASS);             ❺
      System.out.println("Class = " + c);
      Method m = c.getMethod("main", args.getClass());               ❻
      System.out.println("Method descriptor = " + m);
      Object[] passedArgs = { args };
      m.invoke(null, passedArgs);                                    ❼
    } else {
      System.out.println("Compilation failed");
    }
```

```
  }
}
```

❶ The source code that we want to compile. In real life it would probably be dynamically generated, maybe using a `StringBuilder`.

❷ Get a reference to the default `JavaCompiler` object.

❸ Ask the compiler to create a `CompilerTask` to do the compilation. `CompilerTask` is also `Callable` and we save it under that type. The `-d` and `.` are standard `javac` arguments. `MySource` extends the compiler-provided API class `SimpleJavaFileObject` to give access to a file by creating a *file://* URL.

❹ A `Callable` can be put into a thread pool (`ExecutorService`) (see Recipe 11.1); we don't need this capability but the Compiler API returns it. We invoke the `Callable` directly.

❺ Assuming the `result` was `true` indicating success, we load the class with `Class.forName()`.

❻ We have to find the `main()` method in the generated class. We reuse the `String[].class` type from `args`, since all `main` methods have the same argument.

❼ Finally, we can invoke the `main` method, reusing the incoming `args` array to pass any *welcome* message along.

Running this program with and without an argument shows that the argument passed to the `JavaCompilerDemo` is being passed correctly to the generated `AnotherDemo` class:

```
$ java src/main/java/reflection/JavaCompilerDemo.java
Compiled OK
Class = class reflection.AnotherDemo
Method descriptor = public static void
  reflection.AnotherDemo.main(java.lang.String[])
Hi from AnotherDemo
$ java src/main/java/reflection/JavaCompilerDemo.java Welcome
Compiled OK
Class = class reflection.AnotherDemo
Method descriptor = public static void
  reflection.AnotherDemo.main(java.lang.String[])
Welcome from AnotherDemo
$
```

There is a lot to explore in the Compiler API, including the `JavaFileManager` that lets you control the placement of class files (other than by using `-d` as we did here), listeners to monitor compilation, and control of output and error streams. Consult the `javax.tools.JavaCompiler` documentation for details.

17.11 Constructing or Modifying Class Files with the Class-File API 22P

Problem

You want to examine, create, or modify *.class* class files, perhaps in a library, in the tooling of an IDE, in a dynamic interpretation environment, etc.

Solution

Use the new Class-File API, defined in the java.lang.classfile package.

Discussion

The Reflection API has stood us in good stead since before Java 1.0, but there are limits on what it can do. It is primarily used to inspect (or "introspect") and load existing classes. To modify classes, we've had to use third-party libraries such as ASM, a self-proclaimed "all purpose Java bytecode manipulation and analysis framework," which does allow you to create and/or modify Java bytecode. In fact, a copy of ASM has traditionally been bundled as part of the JDK. However, the Java team thought they could do better, and the result is the java.lang.classfile package. Built using modern idioms such as lambdas, this API allows you to:

- Inspect existing class files (when referred to as an array of bytes, not in-memory like the Reflection API)
- Create new class files out of whole cloth—this is intended to be used by compilers, including `javac`
- Modify existing class files at the structural level
- Modify class files at the bytecode or "assembly language" level

For example, aspect-oriented programming (AOP) requires the construction of wrapper classes dynamically at runtime. The new Class-File API is ideal for this. Example 17-15 shows the use of this API to inspect a class file.

Example 17-15. main/src/main/java/classfileapi/ClassFileAPIDemo.java

```
@MyAnnotation
class ClassFileAPIDemo implements Serializable {
  int number = 0;
  void main() throws IOException {
    byte[] classFile = Files.readAllBytes(
      Path.of("target/classes/classfileapi/ClassFileAPIDemo.class"));
    ClassModel model = ClassFile.of().parse(classFile);
    for (ClassElement element : model) {
      switch (element) {
        case FieldModel field ->
          System.out.println(
          "Field " + field.fieldName().stringValue());
        case MethodModel method ->
          System.out.println(
          "Method " + method.methodName().stringValue());
        default -> {
          /* Ignore anything else */
          System.out.println(
          "Other: " + element +
            "(" + element.getClass().getName() + ")");
        }
      }
    }
  }
}
```

The output is fairly voluminous:

```
$ java --enable-preview -cp target/classes classfileapi.ClassFileAPIDemo
Other: AccessFlags[flags=32] (jdk.internal.classfile.impl.AccessFlagsImpl)
Other: ClassFileVersion[majorVersion=66, minorVersion=65535]
        (jdk.internal.classfile.impl.ClassFileVersionImpl)
Other: Superclass[superclassEntry=java/lang/Object]
        (jdk.internal.classfile.impl.SuperclassImpl)
Other: Interfaces[interfaces=java/io/Serializable] (jdk.internal.class-
file.impl.InterfacesImpl)
Field number
Method <init>
Method main
Other: Attribute[name=SourceFile]
        (jdk.internal.classfile.impl.BoundAttribute$BoundSourceFileAttribute)
Other: Attribute[name=RuntimeVisibleAnnotations]
        (jdk.internal.classfile.impl.BoundAttribute$BoundRuntimeVisibleAnnota-
tionsAttribute)
Other: Attribute[name=InnerClasses] (jdk.internal.classfile.impl.BoundAttribute
$BoundInnerClassesAttribute)
$
```

This class examines its own *.class* file, just so we don't have to keep a dummy file around. You'll notice not only the expected `Method` and `Field` listings, but also many

other bits of low-level information, such as the class file version, the source file, the annotation applied to the main class (that's the `RuntimeVisibleAnnotations`), and the definition of the annotation (the last line).

As a longer example, Example 17-16 uses this API to *create a class file entirely within the code*, saves it to disk as a new *.class* file for analysis, and runs its main method.

Example 17-16. main/src/main/java/classfileapi/CreateLoadAndRun.java

```
import static java.lang.constant.ConstantDescs.*;

/**
 * Create a canonical "Hello, World" class using the Class-File API;
 * load it with a ClassLoader, and invoke main() via Reflection.
 * @author Class-File part based on example in Javadoc
 * for java.lang.classfiles
 * @author Glue around that by Ian Darwin
 */
public class CreateLoadAndRun extends ClassLoader {

  final String fullClassName = "notapackage.Hello";    ❶
  // A few ClassDescs that are not in ConstantDescs    ❷
  final ClassDesc CD_fullClassName = ClassDesc.of(fullClassName);
  final ClassDesc CD_System = ClassDesc.of(System.class.getName());
  final ClassDesc CD_PrintStream =
    ClassDesc.of(PrintStream.class.getName());
  // A few MethodDescs that are not in ConstantDescs
  final MethodTypeDesc MTD_void_String =
    MethodTypeDesc.of(CD_void, CD_String);
  final MethodTypeDesc MTD_void_StringArray =
    MethodTypeDesc.of(CD_void, CD_String.arrayType());

  void main() throws Exception {
    byte[] classData = ClassFile
      .of()
      .build(CD_fullClassName,                         ❸
      clb -> clb.withFlags(ClassFile.ACC_PUBLIC)

        // requires no-arg constructor to be valid     ❹
        .withMethod(ConstantDescs.INIT_NAME, ConstantDescs.MTD_void,
          ClassFile.ACC_PUBLIC,
          mb -> mb.withCode(
            cob -> cob.aload(0)                        ❺
              .invokespecial(ConstantDescs.CD_Object,
                ConstantDescs.INIT_NAME, ConstantDescs.MTD_void)
              .return_()))

        // Standard "void main(String[])" main method
        .withMethod("main", MTD_void_StringArray,      ❻
          ClassFile.ACC_PUBLIC + ClassFile.ACC_STATIC,
          methodBuilder -> methodBuilder.withCode(     ❼
```

```
            cob -> cob.getstatic(CD_System, "out", CD_PrintStream)
              .ldc("Hello World")
              .invokevirtual(CD_PrintStream, "println", MTD_void_String)
              .return_())));

    // Save the new class to disk, just so we can examine & verify
    var dirPath = Path.of("/tmp/notapackage");          ❽
    if (!Files.exists(dirPath)) {
      Files.createDirectory(dirPath);
    }
    Files.write(Path.of("/tmp/notapackage/Hello.class"), classData);

    // Define the class in current ClassLoader
    // (e.g., this) and then call main()
    Class<?> c =                                        ❾
      defineClass(fullClassName, classData, 0, classData.length);

    Method m = c.getMethod("main", String[].class);

    m.invoke(null, new Object[]{new String[0]});
  }
}
```

❶ Full class name includes package. Package notapackage is just for humor.

❷ The creators of this tech use the non-Java-standard but reasonable convention of giving `ClassDesc` objects names beginning `CD_` and `MethodTypeDescs` names with `MTD_`. Most of the ones we need are statically imported from `ConstantDescs`.

❸ Start of the call to `build()`; we pass the `CD` for the class name, and a class builder object `clb`, which creates the class as `public`.

❹ Start construcing the no-arg constructor. Every class must have at least one constructor, and a "main program" class typically doesn't need constructors with arguments. `INIT_NAME` is the special name `<init>` used for constructors.

❺ Here we use a code builder `cob` to generate actual machine code ("bytecode") instructions: load a zero value, call the superclass constructor, and return.

❻ This begins the usual `public static void main(String[])` method.

❼ This code builder finds the static field `System.out` of type `PrintStream`, loads the String "Hello World" onto the stack as a parameter, invokes the `PrintStream` method `println()`, and returns.

❽ Straightforward I/O operations. See Chapter 10.

❾ Standard Reflection API. See Recipe 17.4.

This is a very general API and has the ability to transform class files, for example, when a package name changes (as in the Java EE classes all being renamed from "javax." to "jakarta." when Oracle transferred Java EE to the Eclipse Software Foundation). It can do many other types of transformation, such as inserting method calls, removing them, adding or removing individual bytecode files, and more. The reader is referred to the Javadoc for this class; the package-level file contains more details and examples of many transforms.

17.12 Using and Defining Annotations

Problem

You need to know how to use annotations in code or how to define your own annotations.

Solution

Apply annotations in your code using `@AnnotationName` before a class, method, field, etc. Define annotations with `@interface` at the same level as `class`, `interface`, etc.

Discussion

Annotations are a way of adding additional information beyond what the source code conveys. Annotations may be directed at the compiler or at runtime examination. Their syntax was somewhat patterned after Javadoc annotations (such as `@author`, `@version` inside doc comments). Annotations are what I call *class-like things* (so they have initial-cap names) but are prefixed by the `@` sign where used (e.g., `@Override`). You can place them on classes, methods, fields, and a few other places; they must appear immediately before what they annotate (ignoring space and comments). A given annotation may appear only once in a given position (this is relaxed in Java 8 or 9).

As an example of the benefits of a compile-time annotation, consider the common error made when overriding: as shown in Example 17-17, a small error in the method signature can result it an overload when an override was intended.

Example 17-17. MyClass.java (an example of why we need annotations)

```
public class MyClass {

    public boolean equals(MyClass object2) {
        // compare, return boolean
```

```
    }
}
```

The code will compile just fine on any release of Java, but it is *incorrect*. The standard contract of the `equals()` method (see Recipe 8.1) requires a method whose solitary argument is of type `java.lang.Object`. The preceding version creates an accidental overload. Because the main use of `equals()` (and its buddy method `hashCode()`; see Recipe 8.1) is in the `Collections` classes (see Chapter 7), this overloaded method will never get called, resulting in both dead code and incorrect operation of your class within `Sets` and `Maps`.

The solution is very simple: using the annotation `java.lang.Override`, as in Example 17-18, informs the compiler that the annotated method is required to override a method inherited from a supertype (such as a superclass or an interface). If not, the code will not compile.

Example 17-18. MyClass.java with `@Override` annotation

```
public class MyClass {

    @Override
    public boolean equals(MyClass object2) {
        // compare, return boolean
    }
}
```

This version of `equals()`, while still incorrect, will be flagged as erroneous at compile time, potentially avoiding a lot of debugging time. This annotation, on your own classes, will help both at the time you write new code and as you maintain your codebase; if a method is removed from a superclass, all the subclasses that still attempt to override it *and* have the `@Override` annotation will cause an error message, allowing you to remove a bunch of dead code.

The second major use of annotations is to provide metadata at runtime. For example, the Jackson JSON APIs in Recipe 16.2 provide configuration information to its `ObjectMapper`. The Jakarta Persistence API (JPA, see information on my website) uses its own set of annotations from the package javax.persistence to mark up entity classes to be loaded and/or persisted. The Spring Framework lets developers use annotations to provide a range of information

I said that annotations are class-like things, and therefore, you can define your own. The syntax here is a bit funky; you use `@interface`. It is rumored that the team developing this feature was either told not to, or was afraid to, introduce a new keyword into the language, due to the trouble that doing so had caused when the `enum` keyword was introduced in Java SE 1.4. Or maybe they just wanted to use a syntax

that was more reminiscent of the annotation's usage. At any rate, Example 17-19 is a trivial example of a custom annotation.

Example 17-19. Trivial annotation defined

```
package lang;

public @interface MyToyAnnotation {
}
```

Annotations are class-like things, so they should be named the same way—that is, names that begin with a capital letter, and, if public, they are stored in a source file of the same name (e.g, *MyToyAnnotation.java*).

Compile the code in Example 17-19 with `javac` and you'll see there's a new *MyToyAnnotation.class* file. In Example 17-20, we examine this with `javap`, the standard JDK class inspection tool.

Example 17-20. Running `javap` on trivial annotation

```
$ javap lang.MyToyAnnotation
Compiled from "MyToyAnnotation.java"
public interface lang.MyToyAnnotation extends java.lang.annotation.Annotation {
}
$
```

As it says, an `Annotation` is represented in the class file format as just an interface that extends `Annotation` (to answer the obvious question, you could write simple interfaces this way, but it would be a truly terrible idea). In Example 17-21, we take a quick look at `Annotation` itself.

Example 17-21. The `Annotation` interface in detail

```
$ javap java.lang.annotation.Annotation
Compiled from "Annotation.java"
public interface java.lang.annotation.Annotation {
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public abstract java.lang.String toString();
  public abstract java.lang.Class<? extends java.lang.annotation.Annotation>
    annotationType();
}
$
```

Annotations can be made such that the compiler will only allow them in certain points in your code. Example 17-22 is an annotation that can only go on classes or interfaces.

Example 17-22. Sample annotation for classes, interfaces, etc.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
}
```

The `@Target` specifies where the annotation can be used: `ElementType.TYPE` makes it usable on classes, interfaces, class-like things such as enums, even annotations! To restrict it to use on just annotations, there is `ElementType.ANNOTATION_TYPE`. Other types include `METHOD`, `FIELD`, `CONSTRUCTOR`, `LOCAL_VARIABLE`, `PACKAGE`, and `PARAMETER`. So, this annotation is itself annotated with two `@ANNOTATION_TYPE`-targeted annotations.

Usage of annotations with an existing framework requires consulting its documentation. Using annotations for your own purpose at runtime requires use of the Reflection API, as shown in Example 17-23.

One more thing to note about annotations is that they may have attributes. These are defined as methods in the annotation source code but used as attributes where the annotation is used. Example 17-23 is an annotated annotation with one such attribute.

Example 17-23. main/src/main/java/lang/AnnotationDemo.java

```
/**
 * A sample annotation for types (classes, interfaces);
 * it will be available at run time.
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface AnnotationDemo {
  public boolean fancy() default false;
  public int order() default 42;
}

/** A simple example of using the annotation */
@AnnotationDemo(fancy=true)
@Resource(name="Dumbledore")
class FancyClassJustToShowAnnotation {

  /** Print out the annotations attached to this class */
  public static void main(String[] args) {
    Class<?> c = FancyClassJustToShowAnnotation.class;
    System.out.println("Class " + c.getName() + " has these annotations:");
    for (Annotation a : c.getAnnotations()) {
      if (a instanceof AnnotationDemo ad) {
        System.out.println("\t" +a +
          " with fancy=" + ad.fancy() +
```

```
            " and order " + ad.order());
        } else {
          System.out.println("\tSomebody else's annotation: " + a);
        }
      }
    }
}
```

`AnnotationDemo` has the meta-annotation `@Target(ElementType.TYPE)` to indicate that it can annotate user-defined types (such as classes). Other `ElementType` choices include `METHOD`, `FIELD`, and `PARAMETER`. If more than one is needed, use array initializer syntax.

`AnnotationDemo` also has the `@Retention(RetentionPolicy.RUNTIME)` annotation to request that it be preserved until runtime. This is obviously required for any annotation that will be examined by a framework at runtime.

These two meta-annotations are common on user-defined annotations that will be examined at runtime.

The class `FancyClassJustToShowAnnotation` shows the use of `AnnotationDemo` annotation, along with a standard Java one (the `@Resource` annotation). Refer to Recipe 17.13 for a full example of using this mechanism.

17.13 Finding Plug-In-Like Classes via Annotations

Problem

You want to load plug-in classes without using an explicit plug-in API.

Solution

Define an annotation for the purpose, and use it to mark the plug-in classes.

Discussion

Suppose we want to model how the Java EE standard `javax.annotations.Named` or `javax.faces.ManagedBean` annotations work. For each class that is so annotated, convert the class name to an instance-like name (e.g., lowercase the first letter), and do something special with it. You'd want to do something like the following:

1. Get the list of classes in the given package(s) (see Recipe 17.6).
2. Check if the class is annotated.
3. If so, save the name and `Class` descriptor for later use.

This is implemented in Example 17-24.

Example 17-24. main/src/main/java/reflection/PluginsViaAnnotations

```
/** Discover "plug-ins" or other add-in classes via Reflection using Annotations */
public class PluginsViaAnnotations {

  /**
   * Find all classes in the given package which have the given
   * class-level annotation class.
   */
  public static List<Class<?>> findAnnotatedClasses(String packageName,
    Class<? extends Annotation> annotationClass) throws Exception {

    List<Class<?>> ret = new ArrayList<>();
    String[] clazzNames = ClassesInPackage.getPackageContent(packageName);
    for (String clazzName : clazzNames) {
      if (!clazzName.endsWith(".class")) {
        continue;
      }
      clazzName = clazzName.replace('/', '.').replace(".class", "");
      Class<?> c = null;
      try {
        c = Class.forName(clazzName);
      } catch (ClassNotFoundException ex) {
        System.err.println("Weird: class " + clazzName +
          " reported in package but gave CNFE: " + ex);
        continue;
      }
      if (c.isAnnotationPresent(annotationClass) &&
          !ret.contains(c))
          ret.add(c);

    }
    return ret;
  }
```

We can take this one step further and support particular method annotations, similar to `javax.annotations.PostCreate`, which is meant to decorate a method that is to be called after an instance of the bean has been instantiated by the framework. Our flow is now something like this, and the code is shown in Example 17-25:

1. Get the list of classes in the given package(s) (again, see Recipe 17.6).
2. If you are using a class-level annotation, check if the class is annotated.
3. If this class is still of interest, get a list of its methods.
4. For each method, see if it contains a given method-specific annotation.
5. If so, add the class and method to a list of invocable methods.

Example 17-25. main/src/main/java/reflection/PluginsViaAnnotations (find annotated methods)

```
/**
 * Find all classes in the given package which have the given
 * method-level annotation class on at least one method.
 */
public static List<Class<?>> findClassesWithAnnotatedMethods(String packageName,
    Class<? extends Annotation> methodAnnotationClass) throws Exception {
  List<Class<?>> ret = new ArrayList<>();
  String[] clazzNames = ClassesInPackage.getPackageContent(packageName);
  for (String clazzName : clazzNames) {
    if (!clazzName.endsWith(".class")) {
      continue;
    }
    clazzName = clazzName.replace('/', '.').replace(".class", "");
    Class<?> c = null;
    try {
      c = Class.forName(clazzName);
      // System.out.println("Loaded " + c);
    } catch (ClassNotFoundException ex) {
      System.err.println("Weird: class " + clazzName +
        " reported in package but gave CNFE: " + ex);
      continue;
    }
    for (Method m : c.getDeclaredMethods()) {
      // System.out.printf("Class %s Method: %s\n",
      //     c.getSimpleName(), m.getName());
      if (m.isAnnotationPresent(methodAnnotationClass) &&
          !ret.contains(c)) {
        ret.add(c);
      }
    }
  }
  return ret;
}
```

See Also

Recipe 17.12 and the rest of this chapter.

17.14 A Timing Program

Problem

You want to let Java report on the actual time it takes to run a given class.

Solution

Use `System.currentTimeMillis()` before and after invoking the class via the Reflection API.

Discussion

It's pretty easy to build a simplified `time` command in Java, given that you have `System.currentTimeMillis()` to start with. Run my `Time` program, and, on the command line, specify the name of the class to be timed, followed by the arguments (if any) that class needs for running. The program is shown in Example 17-26. The time that the class took is displayed. But remember that `System.currentTimeMillis()` returns clock time, not necessarily CPU time. So you must run it on a machine that isn't running a lot of background processes. Note also that I use dynamic loading (see Recipe 17.1) to let you put the Java class name on the command line.

Example 17-26. main/src/main/java/performance/Time.java

```
public class Time {
  public static void main(String[] argv) throws Exception {
    // Instantiate target class, from argv[0]
    Class<?> c = Class.forName(argv[0]);

    // Find its static main method (use our own argv as the signature).
    Class<?>[] classes = { argv.getClass() };
    Method main = c.getMethod("main", classes);

    // Make new argv array, dropping class name from front.
    // (Normally Java doesn't get the class name, but in
    // this case the user puts the name of the class to time
    // as well as all its arguments...)
    String nargv[] = new String[argv.length - 1];
    System.arraycopy(argv, 1, nargv, 0, nargv.length);

    Object[] nargs = { nargv };

    System.err.println("Starting class " + c);

    // About to start timing run. Important to not do anything
    // (even a println) that would be attributed to the program
    // being timed, from here until we've gotten ending time.

    // Get current (i.e., starting) time
    long t0 = System.currentTimeMillis();

    // Run the main program
    main.invoke(null, nargs);

    // Get ending time, and compute usage
```

```
        long t1 = System.currentTimeMillis();

        long runTime = t1 - t0;

        System.err.println(
            "runTime="  + Double.toString(runTime/1000D));
    }
}
```

Of course, you can't directly compare the results from the operating system `time` command with results from running this program. There is a rather large, but fairly constant, initialization overhead—the JVM startup and the initialization of `Object` and `System.out`, for example—that is included in the former and excluded from the latter. One could even argue that my `Time` program is more accurate because it excludes this constant overhead. But, as noted, it must be run on a single-user machine to yield repeatable results. And no fair watching cat videos in another window while waiting for your timed program to complete!

See Also

Optimizing Cloud Native Java by Benjamin Evans and James Gough (O'Reilly) explains how to tune Java cloud applications for performance "using a quantitative, verifiable, and repeatable approach," addressing various topics that can lead to optimal performance of Java applications running in the cloud. The older *Java Performance* by Charlie Hunt and Binu John (O'Reilly) provides information on tuning Java performance.

17.15 Program: CrossRef

You've probably seen those other Java books that consist entirely of listings of the Java API for version thus-and-such of the JDK. I don't suppose you thought the authors of these works sat down and typed the entire contents from scratch. As a programmer, you would have realized, I hope, that there must be a way to obtain that information from Java. But you might not have realized how easy it is! If you've read this chapter faithfully, you now know that there is one true way: make the computer do the walking. `CrossRef` is a program that puts most of these techniques together. This version generates a cross-reference listing, but by overriding the last few methods, you can easily convert it to print the information in any format you like, including an API reference book. You'd need to deal with the details of this or that publishing software—FrameMaker, troff, TEX, etc.—but that's the easy part.

This program makes fuller use of the Reflection API than did `MyJavaP` in Recipe 17.2. It also uses the `java.util.zip` classes (see Recipe 10.15) to crack the JAR archive containing the class files of the API. Each class file found in the archive is loaded and listed; the listing part is similar to `MyJavaP` (shown in Recipe 17.2).

`CrossRef` is also designed as a base class for subclassing. The methods `startClass()` and `endClass()` are empty. These methods are placeholders designed to make subclassing easy when you need to write something at the start and end of each class. One example is a fancy text-formatting application in which you need to output a bold header at the beginning of each class. Another might be XML, where you want to write `<class>` at the front of each class and `</class>` at the end. `CrossRefXML` is an XML-specific subclass that generates (limited) XML for each field and method.

Both are included in the repository, in the *main/src/main/java/reflection* directory. Both are also in the javasrc GitHub repository, in the *https://github.com/IanDarwin/javasrc/blob/main/main/src/main/java/reflection* directory. By the way, if you publish a book using either of these and get rich, "Remember, remember me!"

See Also

We have not investigated all the ins and outs of reflection or the `ClassLoader` mechanism, but by now you should have a basic idea of how it works. Perhaps the most important omissions are `SecurityManager` and `ProtectionDomain`. The `SecurityManager` was for decades the guardian of access to dangerous APIs. However, it is now deprecated for removal, as in, "terminate with extreme prejudice," and will be removed in a near-future JDK, at the same time as the ancient `Applet` mechanism. A `ProtectionDomain` can be provided with a `ClassLoader` to specify all the permissions needed for the class to run.

I've also left unexplored many topics in the JVM; see the (somewhat dated) O'Reilly books *Java Virtual Machine* by Troy Downing and Jon Meyer, and *Java Language Reference* by Mark Grand. You can also read the Sun/Oracle *Java Language Specification* and *JVM Specification* documents (both updated with new releases, available online), for a lifetime of reading enjoyment and edification!

The Apache Software Foundation maintains a vast array of useful software packages that are free to access and use. The source code is available without charge from the Apache website. Two packages you might want to investigate include the Commons BeanUtils and the Byte Code Engineering Library (BCEL). Commons BeanUtils claims to provide easier-to-use wrappers around some of the Reflection API. BCEL is a third-party toolkit for building and manipulating bytecode class files. Written by Markus Dahm, BCEL has become part of the Apache Commons Project.

CHAPTER 18

Using Java with Other Languages

18.0 Introduction

Java has several methods of running programs written in other languages. You can invoke a compiled program or executable script using `Runtime.exec()`, as I'll describe in Recipe 18.1. There is an element of system dependency here, because you can only run external applications under the operating system they are compiled for. Alternatively, you can invoke one of a number of scripting languages (or *dynamic languages*)—running the gamut: AWK, bsh, Clojure, Ruby, Perl, Python, Scala—using javax.script, as illustrated in Recipe 18.3. Or you can drop down to C level with Java's *native code* mechanism and call between Java and compiled functions written in C/C++; see Recipes 18.6 and 18.7. Java 22 improves on this with the Foreign Function and Memory (FFM) interface, as discussed in Recipe 18.5. From native code, you can call out to functions written in just about any language. Not to mention that you can contact programs written in any language over a socket (see Chapter 15), with HTTP services (see Chapter 15), or with Java clients in RMI or CORBA clients in a variety of languages.

There is a wide range of other JVM languages, including these:

- BeanShell, a general scripting language for Java.
- Groovy, a Java-based scripting language that pioneered the use of closures in the Java language ecosystem. It also has a rapid-development web package called Grails and a build tool called Gradle (see Recipe 2.5).
- Jython, a full Java implementation of Python.
- JRuby, a full Java implementation of the Ruby language.
- Scala, a JVM language that claims to offer the "best of functional and OO" languages.

- Clojure, a predominantly functional Lisp-1 dialect for the JVM.
- Renjin (pronounced "R engine"), a fairly complete open source clone of the R statistics package with the ability to scale to the cloud. See Recipe 12.4 for an example using Renjin.

These are JVM-centric, and some can be called directly from Java to script, or vice versa, without using javax.script. A list of these languages can be found on Wikipedia.

Finally, a few languages provide their own mechanisms for interfacing with Java, such as Perl's `Inline::Java` module to call Java from Perl. These are not covered in *Java Cookbook*. As usual, to go the other way—calling Perl from Java—use javax.script, as in Recipe 18.3.

18.1 Running an External Program from Java

Problem

You want to run an external program from within a Java program.

Solution

Use one of the `exec()` methods in the `java.lang.Runtime` class. Or set up a `ProcessBuilder` and call its `start()` method.

Discussion

The `exec()` method in the `Runtime` class lets you run an external program. The command line you give is broken into strings by a simple `StringTokenizer` (see Recipe 3.1) and passed on to the operating system's "execute a program" system call. As an example, here is a simple program that uses `exec()` to run *kwrite*, a windowed text editor program.[1] On Windows, you'd have to change the name to `notepad` or `wordpad`, possibly including the full pathname, for example, *c:/windows/notepad.exe* (you can also use backslashes, but be careful to double those because the backslash is special in Java strings):

1 *kwrite* is Unix-specific; it's a part of the K Desktop Environment (KDE).

```
public class ExecDemoSimple {

  // Choose one of kate, kwrite, gedit, notepad, wordpad, ...
  static final String EDITOR = "kwrite";

  public static void main(String args[]) throws Exception {

    // Run a "notepad" style editor
    Process p = Runtime.getRuntime().exec(new String[]{EDITOR});

    p.waitFor();
  }
}
```

When you compile and run it, the appropriate editor window appears:

```
$ javac -d . ExecDemoSimple.java
$ java otherlang.ExecDemoSimple # causes a visual editor window to appear.
$
```

This version of `exec()` assumes that the pathname contains no blanks because these break proper operation of the `StringTokenizer`. To overcome this potential problem, use the overload of `exec()`, which takes an array of strings as arguments. Example 18-1 runs the Windows or Unix version of the Firefox web browser, assuming that Firefox was installed in the default directory (or another directory that is on your `PATH`). It passes the name of a help file as an argument, offering a kind of primitive help mechanism, as displayed in Figure 18-1.

Example 18-1. main/src/main/java/otherlang/ExecDemoBrowser.java

```
public class ExecDemoBrowser {
  private static final long serialVersionUID = 1;

  Logger logger = Logger.getLogger(ExecDemoBrowser.class.getName());

  /** The name of the browser. */
  private static final String BROWSER = "firefox";
  /** The name of the help file. */
  protected final static String HELP_FILE = "/help/index.html";
  /** The JFrame */
  protected final JFrame jf;

  /** main - instantiate and run */
  public static void main(String av[]) throws Exception {
    String program = av.length == 0 ? BROWSER : av[0];
    new ExecDemoBrowser(program);
  }

  /** The name of the binary executable that we will run */
  protected static String program;
```

```
/** Constructor - set up strings and things. */
public ExecDemoBrowser(String program) {
  jf = new JFrame("ExecDemoBrowser: " + program + " edition");
  ExecDemoBrowser.program = program;

  Container cp = jf.getContentPane();
  cp.setLayout(new FlowLayout());
  JButton b;
  cp.add(b=new JButton("Exec"));
  b.addActionListener(e -> runProgram());
  cp.add(b=new JButton("Wait"));
  b.addActionListener(e -> doWait());
  cp.add(b=new JButton("Exit"));
  b.addActionListener(e -> System.exit(0));
  jf.pack();
  jf.setLocation(400, 300);
  jf.setVisible(true);
}

Process process;
ExecutorService threadPool = Executors.newFixedThreadPool(1);

/** Start the help, in its own Thread. */
public void runProgram() {

  threadPool.submit(() -> {
      try {
        // Get a "file:" URL for the Help File
        URL helpURL = ExecDemoBrowser.this.getClass().getResource(HELP_FILE);
        if (helpURL == null) {
          JOptionPane.showMessageDialog(jf,
            "Unable to find Help File in resource path",
            "Error",
            JOptionPane.ERROR_MESSAGE);
          System.exit(1);
        }

        // Start the external browser from the Java Application.

        String osName = System.getProperty("os.name");
        String run;
        if ("Mac OS X".equals(osName)) {
          run = "open -a " + program;
          // Any other OSes needing special handling?
        } else {
          run = program;
        }

        process = Runtime.getRuntime().exec(
          new String[]{run, helpURL.toString()});

        logger.info("In main after exec .");
```

```
        } catch (Exception ex) {
          JOptionPane.showMessageDialog(jf,
            "Error" + ex, "Error",
            JOptionPane.ERROR_MESSAGE);
          ex.printStackTrace();  // In terminal window, if any
        }
      });
  }

  public void doWait() {
    if (process == null) {
      logger.info("Nothing to wait for.");
      return;
    }
    logger.info("Waiting for process " + process);
    try {
      process.waitFor();
      // wait for process to complete
      // (may not work as expected for some old Windows programs)
      logger.info("Process " + process + " is done.");
    } catch (Exception ex) {
      JOptionPane.showMessageDialog(jf,
        "Error" + ex, "Error",
        JOptionPane.ERROR_MESSAGE);
    }
  }

}
```

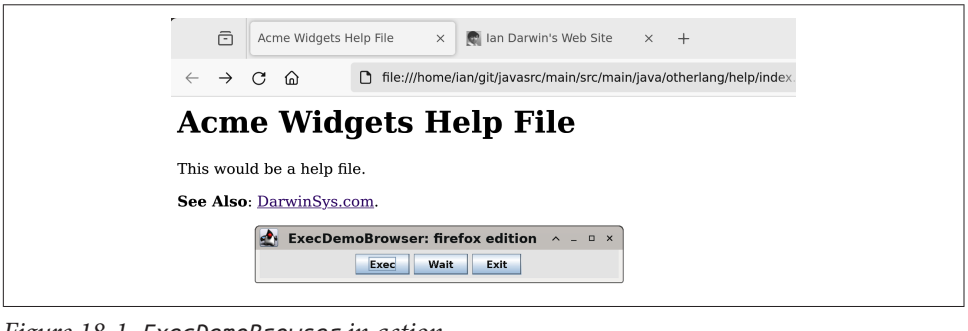


Figure 18-1. `ExecDemoBrowser` in action

Note that this particular example could be replaced by use of the `java.awt.Desktop` class. The Desktop API allows you to open a browser to a specified URL, launch the user's mail client (with an optional mailto URL), or launch the registered application to open/edit/print a file of a type that is known on the underlying operating system. Desktop is not discussed in this book; see the Javadoc or the example *desktop/src/main/java/desktop/DesktopDemo.java*.

A newer class, `ProcessBuilder`, replaces most nontrivial uses of `Runtime.exec()`. `ProcessBuilder` uses generic collections to let you modify or replace the environment, as shown in Example 18-2.

Example 18-2. main/src/main/java/otherlang/ProcessBuilderDemo.java

```
List<String> command = new ArrayList<>();                    ❶
command.add("notepad");
command.add("foo.txt");
ProcessBuilder builder = new ProcessBuilder(command);        ❷
builder.environment().put("PATH",
        "/windows;/windows/system32;/winnt");                ❸
final Process godot = builder.directory(
    new File(System.getProperty("user.home"))).              ❹
    start();
System.err.println("Waiting for Godot");                      ❺
godot.waitFor();                                             ❻
```

❶ Set up the command-line argument list: editor program name and filename.

❷ Use that to start configuring the `ProcessBuilder`.

❸ Configure the builder's environment to a list of common Windows directories.

❹ Set the initial directory to the user's home, and start the process!

❺ I always wanted to be able to use this line in code.

❻ Wait for the end of our little play.

For more on `ProcessBuilder`, see the Javadoc for `java.lang.ProcessBuilder`.

18.2 Running a Program and Capturing Its Output

Problem

You want to run a program but also capture its output.

Solution

Use the `Process` object's `getInputStream()`; read and copy the contents to `System.out` or wherever you want them.

Discussion

The original concept of standard output and standard error was that they would always be connected to the terminal; this dates from an earlier time when almost all computer users worked at the command line. Today, a program's standard output and error output do not always automatically appear anywhere. Arguably, there should be an automatic way to make this happen. But for now, you need to add a few lines of code to grab the program's output and print it:

```
public class ExecDemoLs {

  private static Logger logger =
    Logger.getLogger(ExecDemoLs.class.getSimpleName());

  /** The program to run */
  public static final String PROGRAM = "ls"; // "dir" for Windows
  /** Set to true to end the loop */
  static volatile boolean done = false;

  public static void main(String argv[]) throws IOException {

    final Process p;     // Process tracks one external native process
    BufferedReader is;  // reader for output of process
    String line;

    p = Runtime.getRuntime().exec(new String[]{PROGRAM});

    logger.info("In Main after exec");

    try {
      p.waitFor();
    } catch (InterruptedException ex) {
      // Unlikely, but OK, just quit.
      return;
    }

    // getInputStream gives an Input stream connected to
    // the process p's standard output (and vice versa). We use
    // that to construct a BufferedReader so we can readLine() it.

    is = new BufferedReader(new InputStreamReader(p.getInputStream()));

    while (!done && ((line = is.readLine()) != null))
      System.out.println(line);

    logger.info("In Main after EOF");

    return;
  }
}
```

This is such a common occurrence that I've packaged it up into a class called `ExecAndPrint`, which is part of my com.darwinsys.lang package. `ExecAndPrint` has several overloaded forms of its `run()` method (see the documentation for details), but they all take at least a command and optionally an output file to which the command's output is written. Example 18-3 shows the code for some of these methods.

Example 18-3. darwinsys-api/src/main/java/com/darwinsys/lang/ExecAndPrint.java

```
/** Need a Runtime object for any of these methods */
protected final static Runtime r = Runtime.getRuntime();

/** Run the command given as a String, output to System.out
 * @param cmd The command
 * @return The command's exit status
 * @throws IOException if the command isn't found
 */
public static int run(String cmd) throws IOException {
  return run(cmd, new OutputStreamWriter(System.out));
}

/** Run the command given as a String, output to "out"
 * @param cmd The command and list of arguments
 * @param out The output file
 * @return The command's exit status
 * @throws IOException if the command isn't found
 */
public static int run(String cmd, Writer out) throws IOException {

  Process p = r.exec(cmd);

  FileIO.copyFile(new InputStreamReader(p.getInputStream()), out, true);
  try {
    p.waitFor();  // wait for process to complete
  } catch (InterruptedException e) {
    return -1;
  }
  return p.exitValue();
}
```

As a simple example of using `exec()` directly along with `ExecAndPrint`, I'll create three temporary files, list them (directory listing), and then delete them. The program to do this is shown in Example 18-4.

Example 18-4. main/src/main/java/otherlang/ExecDemoFiles.java

```
// Get and save the Runtime object.
Runtime rt = Runtime.getRuntime();

// Create three temporary files (the slow way!)
```

```
rt.exec(new String[]{"mktemp", "file1"});
rt.exec(new String[]{"mktemp", "file2"});
rt.exec(new String[]{"mktemp", "file3"});

// Run the "ls" (directory lister) program
// with its output sent into a file
String[] args = { "ls", "-l", "file1", "file2", "file3" };
ExecAndPrint.run(args);

rt.exec(new String[]{"rm", "file1", "file2", "file3"});
```

When I run the `ExecDemoFiles` program, it lists the three files it has created:

```
-rw-------  1 ian  wheel  0 Jan 29 14:29 file1
-rw-------  1 ian  wheel  0 Jan 29 14:29 file2
-rw-------  1 ian  wheel  0 Jan 29 14:29 file3
```

A process isn't necessarily destroyed when the Java program that created it—the "parent process"—exits or bombs out. Simple text-based programs will be, but window-based programs like a graphical text editor, browser, or a Java-based `JFrame` or JavaFX application, will not. For example, our `ExecDemoBrowser` program started a web browser, and when `ExecDemoBrowser`'s Exit button is clicked, `ExecDemoBrowser` exits but the browser stays running. What if you want to be sure a process has completed? The `Process` object has a `waitFor()` method that lets you do so, and an `exitValue()` method that tells you the return code from the process. Finally, should you wish to forcibly terminate the other process, you can do so with the `Process` object's `destroy()` method, which takes no argument and returns no value. Example 18-5 is `ExecDemoWait`, a program that runs whatever program you name on the command line (along with arguments), captures the program's standard output, and waits for the program to terminate.

Example 18-5. main/src/main/java/otherlang/ExecDemoWait.java

```
// A Runtime object has methods for dealing with the OS
Runtime r = Runtime.getRuntime();
Process p;     // Process tracks one external native process
BufferedReader is;  // reader for output of process
String line;

// Our argv[0] contains the program to run; remaining elements
// of argv contain args for the target program. This is just
// what is needed for the String[] form of exec.
p = r.exec(argv);

System.out.println("In Main after exec");

// getInputStream gives an Input stream connected to
// the process p's standard output. Just use it to make
// a BufferedReader to readLine() what the program writes out.
```

```
is = new BufferedReader(new InputStreamReader(p.getInputStream()));

while ((line = is.readLine()) != null)
  System.out.println(line);

System.out.println("In Main after EOF");
System.out.flush();
try {
  p.waitFor();  // wait for process to complete
} catch (InterruptedException e) {
  System.err.println(e);  // "Can't Happen"
  return;
}
System.err.println("Process done, exit status was " + p.exitValue());
```

See Also

You wouldn't normally use any form of `exec()` to run one Java program from another in this way; instead, you'd probably create it as a thread within the same process, because this would be faster (the Java interpreter is already up and running, so why wait for another copy of it to start up?). See Chapter 11.

When building industrial-strength applications, note the cautionary remarks in the Java API docs for the `Process` class concerning the danger of losing some of the I/O due to insufficient buffering by the operating system.

18.3 Calling Other Languages via javax.script

Problem

You want to invoke a script written in some other language from within your Java program, running in the JVM, with the ability to pass variables directly to/from the other language.

Solution

If the script you want is written in any of the two-dozen-plus supported languages, use javax.script. Those languages include AWK, Perl, Python, Ruby, BeanShell, Pnuts, Ksh/Bash, R (Renjin), and several implementations of JavaScript.

Discussion

The Scripting Engine mechanism was once quite popular, but it's less so now. One of the first tasks when using this API is to find out the installed scripting engines, and then pick one that is available. The `ScriptEnginesList` program in Example 18-6 lists the engines on the `CLASSPATH` in this project.

Example 18-6. main/src/main/java/otherlang/ScriptEnginesList.java

```
        ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
        System.out.println("Available script engines are: ");
        scriptEngineManager.getEngineFactories().forEach(factory ->
                System.out.println(factory.getLanguageName()));
```

The output was:

```
Available script engines are:
R
SimpleCalc
python
```

As shown, my system at the time of this run had engines named `R`, `SimpleCalc`, and `python`. Conspicuously absent in the output is an engine for ECMAScript (a.k.a. JavaScript). For decades Java would always provide this engine, but no more. Thus the program in Example 18-7 requires the Nashorn JAR file on `CLASSPATH`, which is generously provided by the *pom.xml* file for this project.

Example 18-7. main/src/main/java/otherlang/ScriptEnginesDemo.java

```
public class ScriptEnginesDemo {

  public static void main(String[] args) throws ScriptException {
    ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

    // Run a script in the JavaScript language
    String lang = "JavaScript";
    ScriptEngine engine =
      scriptEngineManager.getEngineByName(lang);
    if (engine == null) {
      System.err.println("Could not find engine");
      return;
    }
    engine.eval("print(\"Hello from " + lang + "\");");
  }
}
```

Example 18-8 is a simple demo of calling Python from Java using `javax.scripting`. We know the name of the scripting engine we want to use: Python. We'll use the JVM implementation known as Jython, which was originally called JPython but was changed due to a trademark issue. Once we add the dependency `org.python:jython-slim:2.7.3` (or later) into the build file, the script engine will be detected when running the program.

Example 18-8. main/src/main/java/otherlang/PythonFromJava.java

```
/**
 * Demo using Python (Jython) to get a Java variable, print, and change it.
 * @author Ian Darwin
 */
public class PythonFromJava {
  private static final String PY_SCRIPTNAME = "pythonfromjava.py";

  public static void main(String[] args) throws Exception {
    ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

    ScriptEngine engine = scriptEngineManager.getEngineByName("python");
    if (engine == null) {
            System.out.println("" +
          "Could not find 'python' engine; add its JAR to CLASSPATH");
      System.exit(1);
    }

    final Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);
    bindings.put("meaning", 42);

    // Let's run a Python script stored on disk (well, on classpath):
    InputStream is =
      PythonFromJava.class.getResourceAsStream("/" + PY_SCRIPTNAME);
    if (is == null) {
      throw new IOException("Could not find file " + PY_SCRIPTNAME);
    }
    engine.eval(new InputStreamReader(is));
    System.out.println("Java: Meaning of life is now " + bindings.get("meaning"));
  }
}
```

Here is the Python script file, from *src/main/resources/pythonfromjava.py*:

```
# Simple Hello in Python
print('Hello from Python')
global meaning
print('Python: Meaning of Life is ' + str(meaning))
# Python doesn't do strong typing:
meaning = 'Good morning'
```

Running the Java program under an IDE yields the following output:

```
Hello from Python
Python: Meaning of Life is 42
Java: Meaning of life is now Good morning
```

The second engine, R, implements the R language as described in Recipe 12.4.

The third engine, the trivial and imaginatively named `SimpleCalc`, is just to show that it is not terribly difficult to build a scripting engine. See my write-up on my website. The file *main/src/main/java/otherlang/SimpleCalcDemo.java* demonstrates my basic `ScriptEngine`, whose source can be found in the *calcscriptengine* subdirectory.

See Also

Before Oracle dismantled *java.net*, there was a list of script engines for many languages (see the archived list; the links don't work, but it shows the extent of the languages that were available). Back then, you could download the script engines from that site. I am not aware of a current official list of engines, nor a download site, unfortunately. However, the list maintained as part of the scripting project can be found in an unofficial source code repository, which I have a copy of in my GitHub repository. It should be possible to build the engine you want from that repository (look in the *engines* subdirectory). A dozen or so other engines are maintained by others outside this project; for example, there is a `Perl5` script engine from Google Code, and an engine that supports compilation of Java(!), available on GitHub.

There is a also a list of Java-compatible scripting languages (not necessarily all using javax.script).

18.4 Mixing Languages with GraalVM 21

Problem

GraalVM aims to be multilanguage, and you'd like to use different languages in the VM.

Solution

Install the current Graal VM as described in Recipe 1.12. Download and use the additional language(s) via Maven or Gradle.

Discussion

While GraalVM positions itself as able to support a wide variety of programming languages, the number currently supported is small but growing. Let's try invoking Python code from within Java. Assuming you've installed Graal itself as per Recipe 1.12, you can add additional languages via Maven or Gradle:

```
    <dependencies>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>polyglot</artifactId>
            <version>${graalvm.version}</version>
        </dependency>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>js</artifactId>
            <version>${graalvm.version}</version>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>python</artifactId>
            <version>${graalvm.version}</version>
            <type>pom</type>
        </dependency>
        <dependency>
            <groupId>org.graalvm.polyglot</groupId>
            <artifactId>tools</artifactId>
            <version>${graalvm.version}</version>
            <type>pom</type>
        </dependency>
    </dependencies>
```

Then the code in Example 18-9 can be built and run, either under the JDK or under Maven. With Maven, the program may appear not to terminate; you can wait about 10 seconds, or use Ctrl+C to interrupt it.

Example 18-9. graal/src/main/java/graaldemo/JavaCallPython.java

```
import org.graalvm.polyglot.Context;
import org.graalvm.polyglot.Value;

/**
 * GraalVM polyglot: calling Python from Java.
 */
public class JavaCallPython {

  public static void main(String[] args)  {

    try (Context context = Context.create("python")) {
      Value result = context.eval("python", "2 + 2");
      int i = result.asInt();
      System.out.println(i);
    }
  }
}
```

This has been run in an IDE and with:

```
mvn exec:java -Dexec.mainClass=graaldemo.JavaCallPython
```

Unsurprisingly, it prints 4. The online source includes a similar program that calls JavaScript from Java. A more complete "starter kit" that also calls Ruby and other languages, and uses a Maven plug-in to build the native image, can be found at the GraalVM GitHub repository.

Graal versions prior to 21 used a command-line tool called `gu` (graal updater) that installed the language packs directly into the JVM, which created maintenance problems. So `gu` was discontinued and replaced by the language packs in Maven Central.

18.5 Calling Between Java and Native Code with the Foreign Function and Memory API 22

Problem

You want easier access to non-Java compiled code.

Solution

Use the Foreign Function and Memory API (FFI/FFM) to connect to functionality provided by compiled libraries in other languages.

Discussion

The Foreign Function and Memory API is described in JEP 454 as a means to access code outside of Java's purview. This was created as part of Project Panama and is often abbreviated to FFI because that's what it's called in many other popular languages, but Java documentation officially refers to it as FFM. The API lets you pass values into C code and retrieve values from the C code. The latter must be compiled, using a C compiler compatible with the one used to build the JDK, and formed into a shared library (`.so`, `.dll`, `.dylib`, depending on the operating system). In addition, you must pay attention to the sizes of values like `int` and `long`, which in C have different sizes on different CPU families.

Example 18-10 is the C code used in this demo. Obviously created by some kind of Monty Python fan, this provides a (very small) subset of the troupe's comedy skits or sketches, and two functions, one for replacing a given sketch title by index number, and another for retrieving the sketch whose title is currently in the given slot. If you don't know C, don't fret too much. It's where a lot of Java's syntax originated, so it

shouldn't be completely opaque, and you can see the two described functions (as well as a `main` function to demonstrate the methods' basic functioning).

Example 18-10. main/src/main/java/ffi/hello-ffi.c

```
/*
 * A silly library just for the FFI demo.
 */

#include <stdio.h>
#include <string.h>

char* sketches[] = {
  "Dead Parrot Sketch",
  "Ministry of Silly Walks",
  "The Funniest Joke In The World",
  "Hell's Grannies"
};

int nSketches = sizeof(sketches)/sizeof(char*);

int updatesketch(int n, char* newVal) {
  if (n > nSketches) {
    fprintf(stderr, "Try to update non-existent sketch!");
    return -1;
  }
  sketches[n] = strdup(newVal);
  printf("Setting %d to %s\n", n, newVal);
  return n;
}

char* pythonsketch(int n) {
  if (n > nSketches)
    return "Very silly";
  return sketches[n];
}

/* A main program, just to show that the function works */
int main(int argc, char **argv) {
  updatesketch(2, "Always look on the bright side of life");
  puts(pythonsketch(2));
}
```

The Makefile is pretty basic, but the arguments passed to the C compiler may need to be changed on other operating systems. The Makefile does this for our Java-calling-C example on my system:

```
    cc -shared -fPIC hello-ffi.c -o hello-ffi.so
```

The Java code has to deal with several concepts and several classes, most in `java.lang.foreign`:

- *Down call* refers to a call from Java down to C level; *up call* refers to a call from native code up to Java (typically a callback).
- Heap memory is allocated by Java's `new` operator—every time you create an object in Java, its storage is in the Java-managed heap. This memory is eligible for garbage collection when no longer in use.
- Off-heap memory is actually allocated in the OS-level heap area (so the term is a bit of a misnomer), typically by system-level functions like `malloc()` in the standard C library. Off-heap memory is never garbage collected by Java. It is thus possible to have memory leaks.
- An `Arena` is responsible for allocating and managing a chunk of off-heap memory; some versions can arrange to be freed up.
- A `MemorySegment` represents a contiguous segment of memory within an `Arena`.
- A `Linker` lets you load shared objects and find functions within them.
- A `FunctionDescriptor` represents the signature of one function in a C library.
- A `MethodHandle` (from `java.lang.invoke`, covered in Recipe 17.5) is a reference to an executable code region (e.g, function).

There are several types of `Arena`s:

- The most common is *confined* (obtained with `Arena.ofConfined()`). These must be closed (commonly via `try-with-resources`) when done, and they are only accessible to the thread that created them.
- A *shared* arena, created with `Arena.ofShared()`, is, as the name implies, shareable across threads, any of which may close it.
- An *automatic* arena, created with `Arena.ofAuto()`, is shareable, and as the name implies, will be deallocated automatically when eligible for garbage collection. It cannot be closed explicitly.
- A *global* arena, created with `Arena.global()`, is shareable and never deallocated. It cannot be closed at all.

With this background, Example 18-11 is a Java program that calls both functions in the C code and displays the results.

Example 18-11. main/src/main/java/ffi/CallCFromJava.java

```
/**
 * Demo program calling two methods in C code, updatesketch() and
 * pythonsketch(). The C code hello-ffi.c has a list of strings that pythonsketch()
 * can retrieve by index number, and an updatesketch() that allows putting
 * a new sketch title into the slot by number. So we update slot #2
 * and then retrieve it to prove that we successfully passed the string
 * in, had it stored by the C code, and that we then retrieved it.
 */
public class CallCFromJava {
  void main() throws Throwable {
    String newSketch = "Killer Bunny";

    try (Arena arena = Arena.ofConfined()) {
      // Allocate off-heap memory and copy 'message' into it
      MemorySegment nativeString = arena.allocateUtf8String(newSketch);
      // Name of C function to find and invoke
      String updateMethod = "updatesketch", getMethod = "pythonsketch";
      // First, get an instance of the native linker
      Linker linker = Linker.nativeLinker();
      // Now get the address of the C function signature
      SymbolLookup ourLib = SymbolLookup.libraryLookup(
        "/home/ian/git/javasrc/main/src/main/java/ffi/hello-ffi.so", arena);
      Optional<MemorySegment> segment = ourLib.find(updateMethod);
      if (segment.isEmpty()) {
        throw new IllegalArgumentException(
          "Method " + updateMethod + " not found!");
      }
      MemorySegment foreignFuncAddr = segment.get();
      // Create argument-list description of the C function
      FunctionDescriptor update_sig=
          FunctionDescriptor.of(ValueLayout.JAVA_LONG, // return
              ValueLayout.JAVA_LONG,  // first argument
              ValueLayout.ADDRESS);   // second
      // Obtain a downcall handle for the C function
      MethodHandle strlen =
        linker.downcallHandle(foreignFuncAddr, update_sig);
      // Finally, what you've all been waiting for:
      //  Call a C function directly from Java
      var ret = (long) strlen.invokeExact(2L, nativeString);
      System.out.println("Update function returned " + ret);

      // Now call getSketch(2) to prove that it got updated
      segment = ourLib.find(getMethod);
      if (segment.isEmpty()) {
        throw new IllegalArgumentException(
          "Method " + getMethod + " not found!");
      }
      foreignFuncAddr = segment.get();
      // Create argument-list description of the C function
      update_sig=
```

```
            FunctionDescriptor.of(ValueLayout.ADDRESS, // return
                ValueLayout.JAVA_LONG);  // only input argument
        // Obtain a downcall handle for the C function
        MethodHandle getter = linker.downcallHandle(foreignFuncAddr,
            update_sig);
        // Once again we call a C function directly from Java
        var retStr = (MemorySegment)getter.invokeExact(2L);
        Consumer<MemorySegment> cleanup = _ -> {
          // Maybe some cleanup here what we had allocated in native space
        };
        retStr = retStr.reinterpret(newSketch.length() + 1, arena, cleanup);
        System.out.println("Update function returned " + retStr);
        String updatedValue = retStr.getUtf8String(0);
        System.out.println("Retrieved updatedValue as " + updatedValue);
      }
    }
}
```

There is quite a bit more to the FFM API, including support for `struct` and `union` types. Refer to the documentation for the java.lang.foreign package and/or to the *FFM Guide* at Oracle, which is Section 12 of the *Java Platform Core Libraries*.

18.6 Calling Other Languages via Native Code (JNI)

Problem

You wish to call native C/C++ functions from Java, either for efficiency or to access hardware- or system-specific features. And you can't use the Java 22+ Foreign Function and Memory API described in Recipe 18.5.

Solution

Use the Java Native Interface (JNI). But note that JNI was superceded in Java 22+ with the Foreign Function and Memory interface described in Recipe 18.5.

Discussion

Java lets you load native or compiled code into your Java program. Why would you want to do such a thing? The best reason would probably be to access OS-dependent functionality or existing code written in another language. A not-as-good reason would be speed: native code can sometimes run faster than Java, though this is becoming less important as computers get faster and more multicore and as the dynamic optimizations of the JVM become more sophisticated.

The native code language bindings are defined for code written in C or C++. If you need to access a language other than C/C++, write a bit of C/C++ and have it pass

control to other functions or applications, using any mechanism defined by your operating system.

Due to such system-dependent features as the interpretation of header files and the allocation of the processor's general-purpose registers, your native code may need to be compiled by the same C compiler used to compile the Java runtime for your platform. For example, on Solaris you can use SunPro C or maybe gcc. On Win32 platforms, use Microsoft Visual C++ version 4.x or higher (32 bit). For Linux and macOS, you should be able to use the provided gcc-based compiler. For other platforms, see your Java vendor's documentation.

Also note that the details in this section are for the JNI of Java 1.1 and later, which differs in some details from 1.0 and from Microsoft's native interface.

Ian's Basic Steps: Java Calling Native Code

To call native code from Java, follow these steps:

1. Write Java code that calls a native method.
2. Compile this Java code.
3. Create an *.h* file using *javah*.
4. Write a C function that does the work.
5. Compile the C code into a loadable object.
6. Try it!

Step 1 is to write Java code that calls a native method. To do this, use the keyword `native` to indicate that the method is native, and provide a static code block that loads your native method using `System.loadLibrary()`. (The dynamically loadable module is created in step 5.) Static blocks are executed when the class containing them is loaded; loading the native code here ensures it is in memory when needed.

Object variables that your native code can modify should carry the `volatile` modifier. The file *HelloJni.java*, shown in Example 18-12, is a good starting point.

Example 18-12. main/src/main/java/jni/HelloJni.java

```
/**
 * A trivial class to show Java Native Interface 1.1 usage from Java.
 */
public class HelloJni {
  int myNumber = 42; // used to show argument passing

  // declare native method
  public native void displayHelloJni();
```

```
  // Application main, call its display method
  public static void main(String[] args) {
    System.out.println("HelloJni starting; args.length="+
                       args.length+"...");
    for (int i=0; i<args.length; i++)
                       System.out.println("args["+i+"]="+args[i]);
    HelloJni hw = new HelloJni();
    hw.displayHelloJni();// call the native function
    System.out.println("Back in Java, \"myNumber\" now " + hw.myNumber);
  }

  // Static code blocks are executed once, when class file is loaded
  static {
    System.loadLibrary("hello");
  }
}
```

Step 2 is simple; just use `javac HelloJni.java` as you normally would. You probably won't get any compilation errors on a basic program like this; if you do, correct them and try the compilation again.

Step 3 requires you to create an *.h* file. On modern versions of Java, this functionality is included in `javac`. The current functionality is used as:

```
$ javac -h output_dir SomeFile.java
```

That is, the `-h` option requires a directory argument (which can be "`.`" but must be specified). This both compiles the Java source into a class file as usual and produces the *.h* file for writing the C code.

In very old versions of Java (up to Java 8), you'd use `javah` to produce this file:

```
$ javah jni.HelloJni           // produces HelloJni.h
```

The *.h* file produced is a glue file, not really meant for human consumption and particularly not for editing. But by inspecting the resulting *.h* file, you'll see that the C method's name is composed of the name `Java`, the package name (if any), the class name, and the method name:

```
JNIEXPORT void JNICALL Java_HelloJni_displayHelloWorld(JNIEnv *env,
    jobject this);
```

Step 4 has you create a C function that does the work. You must use the same function signature as is used in the *.h* file. This function can do whatever it wants. Note that it is passed two arguments: a JVM environment variable and a handle for the `this` object. Table 18-1 shows the correspondence between Java types and the C types (JNI types) used in the C code.

Table 18-1. Java and JNI types

Java type	JNI	Java array type	JNI
byte	jbyte	byte[]	jbyteArray
short	jshort	short[]	jshortArray
int	jint	int[]	jintArray
long	jlong	long[]	jlongArray
float	jfloat	float[]	jfloatArray
double	jdouble	double[]	jdoubleArray
char	jchar	char[]	jcharArray
boolean	jboolean	boolean[]	jbooleanArray
void	jvoid		
Object	jobject	Object[]	jobjectArray
Class	jclass		
String	jstring		
array	jarray		
Throwable	jthrowable		

Example 18-13 is a complete C native implementation. Passed an object of type `HelloJni`, it increments the integer `myNumber` contained in the object.

Example 18-13. main/src/main/java/jni/HelloJni.c

```
#include <jni.h>
#include "jni_HelloJni.h"
#include <stdio.h>
#include <stdlib.h>
/*
 * This is the Java Native implementation of displayHelloJni.
 */
JNIEXPORT void JNICALL Java_jni_HelloJni_displayHelloJni(JNIEnv *env, jobject this)
{
  jfieldID fldid;
  jint n, nn;

  (void)printf("Hello from a Native Method\n");

  if (this == NULL) {
    fprintf(stderr, "'this.' pointer is null!\n");
    return;
  }
  if ((fldid = (*env)->GetFieldID(env,
        (*env)->GetObjectClass(env, this), "myNumber", "I")) == NULL) {
    fprintf(stderr, "GetFieldID failed");
    return;
```

```
  }

  n = (*env)->GetIntField(env, this, fldid);/* retrieve myNumber */
  printf("\"myNumber\" value is %d\n", n);

  (*env)->SetIntField(env, this, fldid, ++n);/* increment it! */
  nn = (*env)->GetIntField(env, this, fldid);

  printf("\"myNumber\" value now %d\n", nn); /* make sure */
  return;
}
```

Step 5 is to compile the C code into a loadable object. Naturally, the details depend on platform, compiler, etc. For example, on Windows, you could use something like this:

```
> set JAVA_HOME=C:\java                 # or wherever
> set INCLUDE=%JAVA_HOME%\include;%JAVA_HOME%\include\Win32;%INCLUDE%
> set LIB=%JAVA_HOME%\lib;%LIB%
> cl HelloJni.c -Fehello.dll -MD -LD
```

And on Unix, you would use something like this:

```
$ export JAVAHOME=/local/java    # or wherever
$ cc -I$JAVAHOME/include -I$JAVAHOME/include/solaris \
        -G HelloJni.c -o libhello.so
```

Example 18-14 is a Unix Makefile that puts all the build steps in one place. You just have to type `make` to build all the pieces, and they are ready to run. The definition of `INCLUDES` needs to be tailored for different locations of the C-language header files.

Example 18-14. main/src/main/java/jni/Makefile (Unix version)

```
# Configuration Section

#CFLAGS_FOR_SO = -G # Solaris
CFLAGS_FOR_SO = -shared
CSRCS    = HelloJni.c
# JAVA_HOME should have been set in the environment
#INCLUDES  = -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/solaris
#INCLUDES  = -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/openbsd
INCLUDES  = -I$(JAVA_HOME)/include -I/Library/Java/JavaVirtualMachines/jdk-22.jdk/
Contents/Home/include/darwin/
LIBDIR=$(JAVA_HOME)/lib/server

all:    testhello testjavafromc

# This part of the Makefile is for C called from Java, in HelloJni
testhello:    hello.all
    @echo
    @echo "Here we test the Java code \"HelloJni\" that calls C code."
    @echo
    LD_LIBRARY_PATH=$(LIBDIR):. java HelloJni.java
```

```
hello.all:    HelloJni.class libhello.so

HelloJni.class: HelloJni.java
    javac -h . HelloJni.java

HelloJni.o::  jni_HelloJni.h

libhello.so:  $(CSRCS) jni_HelloJni.h
  $(CC) -L$(LIBDIR) $(INCLUDES) $(CFLAGS_FOR_SO) $(CSRCS) -o libhello.so

# This part of the Makefile is for Java called from C, in javafromc
testjavafromc:  javafromc.all hello.all
  @echo
  @echo "Now we test HelloJni using javafromc instead of java"
  @echo
  LD_LIBRARY_PATH=$(LIBDIR):. ./javafromc HelloJni
  @echo
  @echo "That was, in case you didn't notice, C->Java->C. And,"
  @echo "incidentally, a replacement for JDK program \"java\" itself!"
  @echo "(not a complete one, nor a very good one, but OK for simple cases)"
  @echo


javafromc.all:  javafromc

javafromc:  javafromc.o
  $(CC) -L$(LIBDIR) -L . javafromc.o -ljvm -o $@

javafromc.o:  javafromc.c
  $(CC) -c $(INCLUDES) javafromc.c

clean:
  rm -f core *.class *.o *.so jni_HelloJni.h
clobber: clean
  rm -f javafromc
```

Step 6: you're done! Just run the Java interpreter on the class file containing the main program. Assuming that you've set whatever system-dependent settings are necessary (possibly including both `CLASSPATH` and `LD_LIBRARY_PATH` or its equivalent), the program should run as follows:

```
    C&gt; java jni.HelloJni
    Hello from a Native Method      // from C
    "myNumber" value is 42          // from C
    "myNumber" value now 43         // from C
    Value of myNumber now 43        // from Java
    C&gt;
```

Congratulations! You've called a native method. However, you've given up portability; the Java class file now requires you to build a loadable object for each operating sys-

tem and hardware platform. Multiply {Windows, macOS, Sun Solaris, HP/UX, Linux, OpenBSD, NetBSD, FreeBSD} times {Intel-32, Intel-64/AMD64, ARM, ARM64, and maybe SPARC64, PowerPC, and HP-PA}, and you begin to see the portability issues.

Beware that problems with your native code can and will crash the runtime process right out from underneath the Java Virtual Machine. The JVM can do nothing to protect itself from poorly written C/C++ code. Memory must be managed by the programmer; there is no automatic garbage collection of memory obtained by the system runtime allocator. You're dealing directly with the operating system and sometimes even the hardware, so be careful. Be very careful.

See Also

If you need more information on Java native methods, you might be interested in the comprehensive treatment found in *Essential JNI: Java Native Interface* by Rob Gordon (Prentice Hall).

18.7 Calling Java from Native Code with JNI

Problem

You need to go the other way, calling Java from C/C++ code.

Solution

Use the Java Native Interface (JNI) again.

Discussion

JNI provides an interface for calling Java from C, with calls to:

1. Create a JVM
2. Load a class
3. Find and call a method from that class (e.g., `main`)

JNI lets you add Java to legacy code. That can be useful for a variety of purposes and lets you treat Java code as an extension language.

The code in Example 18-15 takes a class name from the command line, starts up the JVM, and calls the `main()` method in the class.

Example 18-15. main/src/main/java/jni/javafromc.c (calling Java from C)

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <jni.h>

/*
 * This is a C program that calls Java code.
 * This could be used as a model for building Java into an
 * existing application as an extention language, for example.
 */
int main(int argc, char *argv[]) {
  int i;
  JavaVM *jvm;    /* The Java VM we will use */
  JNIEnv *myEnv;    /* pointer to native environment */
  JavaVMInitArgs jvmArgs; /* JNI initialization arguments */
  jclass myClass, stringClass;  /* pointer to the class type */
  jmethodID myMethod;  /* pointer to the main() method */
  jarray args;    /* becomes an array of Strings */
  jthrowable tossed;  /* Exception object, if we get one. */

  /* initialize the JVM! */
  JavaVMOption options[2];
  options[0].optionString = "-Xmx512m";
  options[1].optionString = "-XX:NonNMethodCodeHeapSize=128m";
  jvmArgs.version = JNI_VERSION_1_6;
  jvmArgs.nOptions = 1;
  jvmArgs.options = options;
  jvmArgs.ignoreUnrecognized = 1;

  if (JNI_CreateJavaVM(&jvm, (void **)&myEnv, &jvmArgs) < 0) {
    fprintf(stderr, "CreateJVM failed\n");
    exit(1);
  }

  /* find the class named in argv[1] */
  if ((myClass = (*myEnv)->FindClass(myEnv, argv[1])) == NULL) {
    fprintf(stderr, "FindClass %s failed\n", argv[1]);
    exit(1);
  }

  /* find the static void main(String[]) method of that class */
  myMethod = (*myEnv)->GetStaticMethodID(
    myEnv, myClass, "main", "([Ljava/lang/String;)V");
  if (myMethod == NULL) {
    fprintf(stderr, "GetStaticMethodID failed\n");
    exit(1);
  }

  /* Since we're calling main, must pass along the command-line arguments,
   * in the form of Java String array */
  if ((stringClass = (*myEnv)->FindClass(myEnv, "java/lang/String")) == NULL){
    fprintf(stderr, "get of String class failed!!\n");
    exit(1);
  }
```

```
  /* make an array of Strings, - 1 for progname & 1 for the class name */
  if ((args = (*myEnv)->
      NewObjectArray(myEnv, argc-2, stringClass, NULL))==NULL) {
    fprintf(stderr, "Create array failed!\n");
    exit(1);
  }

  /* fill the array */
  for (i=2; i<argc; i++)
    (*myEnv)->SetObjectArrayElement(myEnv,
      args, i-2, (*myEnv)->NewStringUTF(myEnv, argv[i]));

  /* finally, call the method. */
  (*myEnv)->CallStaticVoidMethodA(myEnv, myClass,
    myMethod, (const jvalue *)&args);

  /* And check for exceptions */
  if ((tossed = (*myEnv)->ExceptionOccurred(myEnv)) != NULL) {
    fprintf(stderr, "%s: Exception detected:\n", argv[0]);
    (*myEnv)->ExceptionDescribe(myEnv);  /* writes on stderr */
    (*myEnv)->ExceptionClear(myEnv);  /* We're done with it. */
  }

  (*jvm)->DestroyJavaVM(jvm); /* no error checking as we're done anyhow */
  return 0;
}
```

Afterword

Writing this book—and keeping it up to date—has been a humbling experience. It has taken far longer than I had predicted or than I would like to admit. And, of course, it's not finished yet. Despite my best efforts and those of the technical reviewers, editors, and many other talented folks, a book this size is bound to contain errors, omissions, and passages that are less clear than they might be. Do let us know if you happen across any of these things; you can view and submit errata through the book's catalog page. Subsequent editions will incorporate changes sent in by readers just like you!

It has been said that you don't really know something until you've taught it. I have found this true of lecturing, and I find it equally true of writing.

I tell my students that when Java was very young, it was possible for one person to study hard and know almost everything about it. After a release or two, this was no longer true. Today, nobody in their right mind would seriously claim to "know all about Java"—if they do, it should cause your bogosity detector to go off at full volume. And the amount you need to know keeps growing. How can you keep up? Java books? Java magazines? Java courses? Conferences? There is no single answer; all of these are useful to some people. Oracle and others have programs that you should be aware of:

- For many years, JavaOne was the dominant conference on Java, put on by Sun Microsystems and briefly by Oracle, then shuttered for a few years. Recently, Oracle has folded this into CloudWorld, the annual Oracle conference. More recently, Oracle announced that JavaOne would be making a comeback in March 2025 in California. Visit the JavaOne website for the latest on this.
- There is a sporadically updated list of Java converences at JavaConferences.org.
- The Oracle Java Technology Network is a free web-based service for getting the latest APIs, news, and views.

- Over Java's lifetime, the publishing industry has changed a lot. There used to be several Java-related magazines published in print, some of whose articles would appear on the web. Today there are, so far as I know, no print magazines dedicated to Java. Oracle until recently (2023) published the online-only *Java Magazine* every month with technical articles on many aspects of Java written by professionals in tech who were also good writers (immodest to say that, as I was one of those writers); see the magazine's website to view past issues. Now they are using Oracle staff for many articles, and "accepting" community-contributed articles.
- The Java Community Process is the home of Java standardization and enhancement.
- The OpenJDK community maintains and builds the open source version of the "official" JDK.
- O'Reilly books and conferences are among the very best available!
- I keep my own list of Java resources that I update sporadically, on my Java site; follow the link to my Java Resources List.
- A good source of Java news is Ken Kousen's Tales from the Jar Side newsletter.
- Some very interesting advanced topic discussions show up in Heinz Kabutz's Java Specialists' Newsletter.

There is no end of Java APIs to learn about. And there are still more books to be written . . . and read.

APPENDIX A

Java Then and Now

Introduction: Always in Motion the Java Is

Java has always been a moving target for developers and writers. This I know as I've been updating, first a Java training program and later this book, for pretty much all of Java's lifetime. Some developers I meet in my commercial training programs are still not aware of some of the features added to ancient Java releases, let alone current ones. This appendix offers a look at each recent release of Java. Details on releases prior to Java 16 are considered ancient history and have been moved to my website, *https://darwinsys.com/java/ancientHistory.html*. For a review of the very early history, see Jon Byous's Sun Microsystems article "Java Technology: The Early Years".

There are many references to "JEP" (Java Enhancement Proposal) in this section. To get more information on any of these, simply visit *openjdk.org/jeps/NNN* where *NNN* is the JEP number.

What Was New in Java 16 16

The biggest items in Java are the `record` type and the `jpackage` tool, but there are many more.

Another big change, not in the language but in how the JDK source code is maintained, is the move from half a dozen separate repositories in Mercurial, a smaller competitor to Git, into a unified repository on GitHub. These changes were introduced in JEP 357: Migrate from Mercurial to Git and JEP 369: Migrate to GitHub, and I wrote about them in Oracle's *Java Magazine*.

Other changes of note include supported ports of the JDK for JEP 386: Alpine Linux (one of the smallest mainstream Linux distros, often used in cloud deployments) and JEP 388: Windows on AArch64 (64-bit ARM processors).

Java 16 Language Changes

The biggest single change is probably the addition of the `record` type (JEP 395, previously in preview, and covered in Recipe 8.4). The `record` type allows the rapid production of an immutable data holder object, with all getters and ancilliary methods provided.

Another change is pattern matching with `instanceof` (JEP 394). This allows you to replace:

```
if (x instance of MyClass) {
        MyClass m = (MyClass)x;
        // do something with m
}
```

with just:

```
if (x instance of MyClass m) {
        // do something with m, no cast needed!
}
```

There is more on pattern matching in later releases of Java, as this is just the first step toward a language feature called pattern matching.

JEP 396 marks the primitive wrapper classes as value-based and deprecates their constructors (only) for removal. It also warns about improper attempts to synchronize (see Recipe 11.4) on an instance of any JDK value-based class. (There is a private annotation `jdk.internal.ValueBased` used internally and recognized by `javac`.)

Sealed classes (JEP 397, see Recipe 8.11) continued as a preview feature in Java 16.

Java 16 API Changes

You can simply collect a `Stream` into a `List`: the terminal operation `collect(Collectors.toList())` can be replaced with just `toList()`. Be aware that this does produce an immutable list.

JEP 396 tightens the encapsulation of JDK internal classes to include all but a very few, such as the notorious-but-necessary `Unsafe` class.

Incubation features include the Vector API (JEP 338, covered in Recipe 5.11), the Foreign Linker API, and the Foreign Function and Memory (FFI/FFM) API.

Unix domain sockets JEP 380, a facility in Unix since 1980, was added to Microsoft Windows, allowing for it to be supported in Java 16. See Recipe 15.8.

One format character is added to `DateTimeFormatter` (Recipe 6.2): `B` allows a more verbose period instead of just a.m. or p.m., such as "in the afternoon."

Java 16 JDK Tools Changes

The `jpackage` tool (JEP 392: Packaging Tool, covered in Recipe 2.18) provides a means of packaging a Java application along with all the APIs that it needs *and* the subset of the JDK that is needed to run it. Works on Windows, Linux, and macOS.

Java 16 JVM Changes

Improvements to garbage collection (GC) include the release of class metadata memory and the JEP 439: Generational ZGC.

What Was New in Java 17 LTS 17

Java 17 was the next long-term supported release and incorporated several useful features.

Java 17 Language Changes

JEP 409 Sealed classes (Recipe 8.11) move from preview to final.

Java 17 API Changes

The random number functionality is significantly expanded, as per JEP 356. I've written this up in a two-part series in Oracle's *Java Magazine*.

Java 17 switches to strict floating-point calculations by default (JEP 306), so the keyword `strictfp` is now optional and ignored; all FP calculations are done in strict conformance to the IEEE 754 floating-point standard.

Better rendering on macOS (JEP 382), and initial support for macOS AArch64 (Apple's ARM 64 M1, M2, … CPUs) are in JEP 391.

JEP 406 introduces pattern matching for `switch` in preview—as mentioned earlier, pattern matching is coming to several parts of Java syntax.

JEP 412 Foreign Function and Memory API (Incubator) to allow access to native code and memory. See Recipe 18.5.

JEP 414 Vector API (Second Incubator)—for faster numerical operations on arrays. See Recipe 5.11.

JEP 403 Strongly Encapsulate JDK Internals—makes it even harder to access JDK internal APIs that you shouldn't be accessing anyway :-)

The obsolete Security Manager (JEP 411) and the long-dead Applet API (JEP 398) are deprecated for removal—both are done and it's time for them to go.

The Experimental AOT and JIT compilers were removed in JEP 410.

The Activation mechanism of RMI was removed in JEP 407.

What Was New in Java 18 18

`finalize()` finally finding its fate, and a free web server, are two of the biggest changes.

Java 18 Language Changes

The `finalize()` method has finally been deprecated for removal in JEP 421. It was error-prone from the beginning of time, and experts have advised against its use since forever. It will be gone soon, and that's `final`.

Java 18 API Changes

The default charset is now UTF-8 on all platforms thanks to JEP 400, allowing for better file interchange between Java on different operating systems.

A simple web server for static files has finally been added via JEP 408 (Python has had such a thing for a dog's age). This is invoked via the `jwebserver` command-line tool. It can also be started using this *unsupported* incantation; don't blame me if this changes in a future release, though I've tested it up to Java 24:

```
java --add-modules=jdk.httpserver sun.net.httpserver.simpleserver.JWebServer
```

In JEP 416, some core parts of the Reflection API (Chapter 17) have been reimplemented on top of the `MethodHandles` API (Recipe 17.5) to reduce code duplication inside the JDK.

Pattern matching for `switch` is in its second preview (JEP 420).

The Vector API (Recipe 5.11) is in its third incubator.

The Foreign Function and Memory API (Recipe 18.5) is in its second incubator.

JEP 418 provides a service-provider interface (SPI) for DNS host resolution, so the `java.net.InetAddress` class can use resolvers other than the platform's built-in resolver.

Java 18 Tooling Changes

In JEP 413, Javadoc gains the ability to include code snippets (see "Javadoc Snippets"), meant to illustrate how to use a given class.

What Was New in Java 19 19

Java 19 continues the steady evolution of Java.

Java 19 Language Changes

Record patterns in preview (JEP 405) allows nesting of record patterns and type patterns. This extends pattern matching to allow for more sophisticated and composable data queries.

Pattern matching for `switch` (JEP 427), in its third preview, allows pattern matching in `switch` expressions and statements, permitting an expression to be checked against multiple patterns.

These two together are significant steps toward what's being called data-oriented programming (see Recipe 8.15).

Java 19 API Changes

Foreign Function and Memory API (Recipe 18.5) moves from incubator to preview (JEP 424).

The Vector API (Recipe 5.11) is back, now in its fourth incubator (JEP 426).

Virtual threads (Recipe 11.2) appears in its first preview (JEP 425).

Structured concurrency (JEP 428, see Recipe 11.7) is here as an incubator feature.

Java 19 JVM Changes

A port to the RISC/V processor on Linux, via JEP 422.

What Was New in Java 20 20

Nothing radically new here—all but one of the major enhancements are continued preview or incubator features.

Java 20 Language Changes

Record patterns appears in its second preview (JEP 432).

Pattern matching for `switch` in its fourth preview (JEP 433).

Java 20 API Changes

Scoped values (JEP 429, described in Recipe 11.7) is new, as an incubator feature.

Virtual threads (JEP 436, covered in Recipe 11.2) is in its second preview.

Structured concurrency (JEP 437, see Recipe 11.7) is in its second incubator.

Foreign Function and Memory API (JEP 434, described in Recipe 18.5) continues as a second preview.

The Vector API (JEP 438, covered in Recipe 5.11) is in its fifth incubator release.

What Was New in Java 21 LTS 21

Java 21 is the latest and greatest LTS as of this writing, but, of course, not for long, as new releases appear every six months. It will be the current LTS release until, we believe, 2025.

Java 21 Language Changes

Record patterns in `switch` (JEP 440) and pattern matching for `switch` (JEP 441) together have been modified to allow for simpler use of `switch` statements and record unpacking; see Recipe 8.9.

JEP 443 in preview offers unnamed patterns and variables—using _ for the name of an argument that is never going to be used.

Java 21 API Changes

Virtual threads (JEP 444) allow for significant performance improvement of non-CPU-bound applications (see Recipe 11.2).

Sequenced collections (JEP 431) allow navigation of collections with first, last, and next element retrieval.

JEP 452 offers a new Key Encapsulation Mechanism API (KEMs can be used in cryptography to pass keys securely).

Java 21 JVM Changes

Generational ZGC (JEP 439)—the ZGC garbage collector now uses both an old and a new space, which aims to improve performance beyond what ZGC brought previously.

JEP 449 deprecates the Windows 32-bit x86 port for removal.

JEP 451 prepares to to disallow dynamic loading of monitoring agents.

Java 21 Preview Changes

One of the biggest changes is unnamed classes and instance main (JEP 445, see Recipe 1.2).

Another biggie was the ill-fated string interpolation (JEP 430).

Structured concurrency (JEP 453) and scoped values (JEP 446) provide better support for concurrent applications.

The Foreign Function and Memory API (JEP 442) allows for easier access to native code and the ability to share memory with a native function.

Java 21 Incubator Changes

The Vector API (JEP 448, see Recipe 5.11) has minor improvements.

What Was New in Java 22 22

Java 22 Language Changes

JEP 456, Unnamed Variables & Patterns, now out of preview, allows use of _ as a variable name in many contexts to signify that the variable is not used, as in this Swing button handler:

```
quitButton.addActionListener(_ -> System.exit(1));
```

The argument (which is of type `ActionListener`) would normally have been declared with a name like `e` or `evt`, but is not actually used in the lambda; the _ optimization tells the compiler explicitly to disregard it.

Java 22 API Changes

JEP 454, the Foreign Function and Memory API, is out of preview (see Recipe 18.5).

JEP 457 provides a preview of the Class-File API (see Recipe 17.11).

Java 22 Tooling Changes

`javac` is extended by JEP 458 to allow you to launch multi-file source-code programs.

Java 22 Preview Changes

Statements before `super(...)` (JEP 447) allows statements that don't reference current-object fields to appear before a call to `super()`, allowing simplification of error checking, and otherwise relaxes the strictness; see Recipe 8.2.

JEP 461: Stream Gatherers enhances the Streams API (Recipe 9.3) to provide for easier coding of intermediate operators. Covered in Recipe 9.5.

Following remain in preview:

- JEP 459: String Templates or String Interpolation remained a preview feature in Java 22 (but was sadly killed off in Java 23).
- JEP 462: Structured Concurrency (Second Preview)

- JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)
- JEP 464: Scoped Values (Second Preview)

Java 22 Incubator Changes

The Vector API (Recipe 5.11) reaches a record-setting seventh incubation, with only minor changes.

What's New in Java 23 23

This is the latest release as this book is going to press for its fifth edition.

Java 23 language Changes

JEP 471 deprecates for removal the memory-access methods in `sun.misc.Unsafe`.

Implicitly declared classes and instance main methods (JEP 477) are in their third preview.

JEP 455 offers the initial preview of primitive types in patterns, `instanceof`, and `switch`.

Module import declarations (JEP 476, in preview) allow you to import all the exports from a given module with one statement: `import module` *`modulename`*.

Java 23 API Changes

The Vector API is in its eighth(!) incubation, as JEP 469.

Stream gatherers (JEP 473) and the Class-File API (JEP 466), both in their second preview.

Java 23 JVM Changes

JEP 474 makes the generational mode of the ZGC garbage collector the default mode.

Java 23 Tooling Changes

JEP 467: Javadoc now accepts Markdown documentation comments; see "Markdown 23" on page 28 for an example.

Java 23 Preview

String templates preview is *removed*, which I don't think has happened before, or at least very rarely.

The following remain in preview:

- JEP 480: Structured Concurrency (Third Preview)
- JEP 481: Scoped Values (Third Preview)
- JEP 482: Flexible Constructor Bodies (Second Preview)

What's New in Java 24 24

Java 24 will be released within a week or so of this book's release. The following JEPs have been announced for it, and this is claimed to be the final list.

Language Changes

- JEP 472: Prepare to Restrict the Use of JNI (not to deprecate, but to require explicit enabling of risky native code)
- JEP 488: Primitive Types in Patterns, `instanceof`, and `switch` (Second Preview)
- JEP 492: Flexible Constructor Bodies (Third Preview)
- JEP 494: Module Import Declarations (Second Preview)
- JEP 495: Simple Source Files and Instance Main Methods (Fourth Preview)

API Changes

- JEP 478: Key Derivation Function API (Preview)
- JEP 484: Class-File API (out of preview)
- JEP 485: Stream Gatherers (out of preview)
- JEP 486: Permanently Disable the Security Manager
- JEP 487: Scoped Values (Fourth Preview)
- JEP 489: Vector API (in a record-setting ninth incubator)
- JEP 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe
- JEP 499: Structured Concurrency (Fourth Preview)
- JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism
- JEP 497: Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm

GC Changes

- JEP 404: Generational Shenandoah (Experimental)
- JEP 475: Late Barrier Expansion for G1 Garbage Collection

- JEP 490: ZGC: Remove the Non-Generational Mode

JVM Changes

- JEP 450: Compact Object Headers (Experimental)
- JEP 479: Remove the Windows 32-bit x86 Port
- JEP 483: Ahead-of-Time Class Loading & Linking
- JEP 491: Synchronize Virtual Threads without Pinning
- JEP 493: Linking Run-Time Images without JMODs
- JEP 501: Deprecate the 32-bit x86 Port for Removal

Looking Ahead

> Nobody's armchair is a good predicter of the future.
>
> —Mike O'Dell

One thing we know is that there will be a new release of Java every six months, off into the indeterminate and unknowable future. I hope that string templates will come back to life, and that most of the other preview and incubator features in recent releases will become standard.

Index

Symbols

" (tar command), 405
$ (dollar sign)
  newline matching, 174
  R language, 466
% (percent symbol), printf formatting, 224
() (parentheses), regex grouping, 163
+ (plus operator), string concatenation, 121
-> (arrow notation), lambda expression, 195, 332
-x (extract) switch, 405
. (dot)
  JAR file creating, 32
  text formatting commands, 146
/ (slash)
  filename separator, 46-47
  JSON Pointers, 548
/// (triple slash), Markdown, 28
:: (double colon), method reference, 341, 348
< > (diamond operator), 249
<- (assignment arrow), 460
== (equals operator), 288
@ (annotations), 30, 577
[ ] (square brackets), Markdown, 28
\ (escape character), 140, 153
\r\n (end-of-line characters), 403, 520
^ (carat) character, 174
{ } (braces), Javadoc code snippets, 26

A

abstract class, 300
abstract methods, 304-305
AbstractStringBuilder class, 122
accented characters, matching, 172
accept() method, 515, 519, 523
acceptor-type methods, 335
accumulator() function, 340
ActionListener interface, 330, 333
actionPerformed() method, 328
AD versus BC years in date/time pattern, 225
ADAMS, 471
add() method
  List, 250
  Set, 260
  simple array, 433
  Vector, 250
addAll() method, 261
addElement() method, 250
addresses, network, 497-500
Advanced Vector Extensions (AVX), 204
AGI (Artificial General Intelligence), 469
AI (artificial intelligence), 455
  (see also machine learning)
AI Service interface, 480-482
Akka-Streams, 356
Alexander, Christopher, 283
aligning strings, 126-129
alternatives command, 40
Android Cookbook (Darwin), xix
Annotation interface, 579
annotation processor, 299
annotations, 30-31
  @Deprecated, 74
  @FunctionalInterface, 337
  @interface, 577, 578
  marking plug-in classes, 581-583
  @Override, 289, 578
  @Retention, 581

@Target, 580, 581
@Test, 78
using with reflection, 577-581
anonymous inner classes, 332
anonymous methods, 330
anonymous package, 51
ANTLR, 391
AOP (aspect-oriented programming), 285, 573
Apache Commons BeanUtils, 586
Apache Commons Logging, 83
Apache Commons Math library, 207
Apache Commons Text, 139
Apache Hadoop, 455
Apache HttpClient, 496
Apache Logging Services Project's Log4j, 83, 86-91
Apache Maven
automating with, 61-66
and CLASSPATH, 23, 60
dependency management, 68-71
downloading code examples, 7, 10
versus Gradle, 59
JAR file creation, 33-35
javasrc library modules, 8
versus JPMS modules, 52
JUnit with, 79
mvn archetype:generate command, 63
mvn package, 38
mvn versions command, 70
pros and cons, 64
and R, 463
running Spark with mvn, 458
web archive file creation, 38
Apache Software Foundation, 586
Apache Spark, 455, 456-459
Apache Subversion, 11
Apache Tomcat, 526
append() method, 103, 121, 122, 130
Appender (Logger), 88
appendReplacement() method, 166
appendTail() method, 166
Apple Xcode, 17
applets, 552
Application class, 532-534
apply() method, 337
applyAsInt() method, 337
applyPattern() method, 191
Arena, 603
global() method, 603
ofAuto() method, 603
ofConfined() method, 603
ofShared() method, 603
arithmetic operations on large number collection, 203-207
arithmetic operators, Vector class, 207
ArrayIndexOutOfBoundsException, 242, 260
ArrayList class, 248-249, 284
arrays, 239
(see also Vector API)
converting collections to, 276
creating to hold data, 241-242, 280-282
end-of-line character issue, 404
lack of in Java strings, 112
multiplying matrices, 201-203
resizing, 242-243
Arrays class, 240, 243-245
Arrays.asList() method, 250
Arrays.binarySearch() method, 275
Arrays.copyOf(), 284
Arrays.sort() method, 268-270, 275
Arrays.stream() method, 338
Arrays.toString() method, 244
arrow notation (->), lambda expression, 332
Artificial General Intelligence (AGI), 469
artificial intelligence (AI), 455
(see also machine learning)
aspect-oriented programming (AOP), 285, 573
assert keyword, versus JUnit, 80
AssertJ, 80
assertThat() method, 79
associative data structure type, 240
asynchronous sockets, 492
at character (@) for annotations, 30
atomic variables, 422
attributes
annotations, 580
assigning, 376-379
atZone() method, 235
auto-boxing/auto-unboxing, 182
auto-vectorization, 204
AutoClosable interface, 440
automatic arena, 603
automatic modules, 57
automating build process (see Apache Maven; Gradle)
AVX (Advanced Vector Extensions), 204
Azul Platform Prime, 3

B

back pressure, data control, 354
backslash (\) character, 140, 153
Bakker, Paul, 52
batch refactoring, 75-77
BC versus AD years in date/time pattern, 225
BCEL (Byte Code Engineering Library) (Dahm), 586
BeanShell, 587
Beck, Kent, 77
bias issue for AI, 474
Big Data (see data science)
big numbers, 179
BigDecimal class, 210
BigInteger class, 210
binary data
  client-side transfer, 504-507
  as input/output, 379-381, 383
  server-side returning of response, 518-522
binary numbers, 194
binarySearch() method, 274, 276
bison parser generator, 391
BitSet class, 240
Bloch, Joshua, 284, 308
BlockingQueue subinterface, 444-446
boilerplate code, managing, 297-299
boolean flag variable, 123
boolean methods (Files), 363-364
boolean variables, 160, 424
bridging, 252
Brown, Doug, 391
BufferedInputStream() method, 393
BufferedReader subclass
  files, 381, 382
  standard input, reading from, 393, 394
  strings, 110, 174
  textual data reading, 501
BufferedReader.lines() method, 395
build tools, 96, 588, 592
  (see also StringBuilder)
Burke, Bill, 496
Byte Code Engineering Library (BCEL), 586
byte[ ] Files.readAllBytes method, 381

C

C languages
  in FFI/FFM API, 601-605
  with Java Native Interface, 605-611
  numeric types versus Java, 177
  socket synchronization challenge, 492
  strings versus Java, 112, 116
  unsafe enumerations, 308
Calendar class, 215, 235-237
call() method, 423
Callable interface, 423, 430, 571
callback functions, and closures, 331
CameraAcceptor interface, 335
cancel() method, 427
canonical equivalence, 171
canonical matching, 172
canonical name, 368
CANON_EQ flag, 171, 172
capitalization of names, 139
capitalize() method, 139
capitalizeFully() method, 139
capture groups, regex, 163
carat (^) character, 174
case, controlling, 138-139, 171-172
CASE_INSENSITIVE flag, 171, 172
Cassandra database, 455
CDI (Contexts and Dependency Injection), 321, 514
certificate management (SSL), 530-532
chained method calls, 338
chaining operations, 428
Chambers, John, 462
char data type, 124, 130
character strings (see strings)
Character.isSpace() method, 115
CharacterIterator interface, 168
characters
  adding nonprintable, 140-141
  converting sets, 394
  matching accented, 172
  processing one at a time, 124
  reversing in string, 132
  Unicode, 129-131, 152, 380, 401
charAt() method, 124, 130
chat client, sockets-based, 511
ChatGPT, 470, 476-480
chatGPT() method, 478
ChatGptRequest, 478
ChatMemory, 483
checked exceptions, 320
checksum, 125
Cheswick, William R., 538
ChoiceFormat class, 197-198
choose() method, 333

ChronoUnit class, 230
CI (continuous integration), 95-100
Class class, 409, 551, 567
  (see also reflection)
class equality, 290
class keyword, 556
Class-File API, 573-577
Class.forName() method, 45, 552, 553
classes
  creating in IDE, 15
  dependency injection for, 321-323
  file construction or modification, 573-577
  inner classes, accessing, 565-567
  JavaCompiler, constructing with, 570-573
  listing, in packages, 563-565
  loading/instantiating dynamically, 552-554
  plug-in-like, 581-583
  printing information of, 554-555
  sealed, 315-317
  testing environment for, 45
  threading (see threading)
ClassesInPackage class, 563
classification, in machine learning, 469
classless main, 4-7, 368
ClassLoader class, 409, 551, 568-570, 586
CLASSPATH, 3
  and Maven and Gradle, 23, 60
  relative versus absolute resource names, 410
  scanning for classes in, 563-565
  setting to directory or JAR file list, 21-23
clean command, 63
cleanup methods, when freeing objects, 284
client-side network, 491-512
  address finding and reporting, 497-500
  binary data transfer, 504-507
  chat client, sockets-based, 511
  contacting socket server, 496-497
  error handling, 500-501
  HTTP/REST web client, 494-496
  input/output and socket, 518-520
  Java API for ChatGPT, 476
  layering of networks, 491
  multiple-client handling by server, 522-525
  read-and-write model in, 357
  text data transfer, 501-503
  UDP datagrams, 507-509
  URI, URL, and URN, 510-511
Clock class, 219
Clock.fixed() method, 219
Clojure, 328, 588
clone() method, 284, 293
close() method, 443
closures, using lambdas to replicate, 331
code completion, IDE feature, 13
code examples, xxii, 7-12
code folding, 333
code management, CI as tool for, 96
code reuse, informing developers of, 23-25
codePointAt() method, 124, 130
coding AI, 469
Colebourne, Stephen, 215
collect() method, 342
Collection interface, 240, 274-276
collections (see streams)
Collections API, 251
Collections Framework, 240, 246-247
Collections interface/classes, 240
  converting to arrays, 276
  creating multidimensional structures, 281
  finding objects in, 274-276
  List class, 246, 247-252
  Singleton methods, 320
  sorting, 268-273
Collections.binarySearch() method, 276
Collections.sort() method, 268
Collector terminal function, simplifying streams with, 339-342
Collectors class, 341
com.darwinsys packages, 9
combiner() function, 340, 344
command-line JDK, 2
COMMENTS flag (regex), 172
comments, Javadoc, 23-25
CompactNumberFormat, 192
Comparable interface, 270-272
Comparator interface, 268-273
Comparator.comparing() method, 269
compare() method, 244
compareTo() method, 270
compareToIgnoreCase() method, 268, 351
compareUnsigned() methods, 245
compile command, 63
compile time, testing code for version or OS, 46
compiler, packaging rules, 51
compiler/IDE warning messages, 75-77
CompilerTask object, 571
CompletableFuture class, 422, 428-429
complete() method, 428

Complex class, 207-209
complex numbers, 207-209
composite characters, matching, 172
compressing tabs, 133-137
compute() method, 447
computeArea() method, 301
computeReplacement method, 167
concurrency
  Fork/Join pool in alternative models, 450
  structuring threads, 440-444
  utilities, 422
Concurrent Versions System (CVS), 11
ConcurrentModificationException, 263
Condition objects, 440
conditional expressions, number formatting, 197-198
confined arena, 603
connect() method, 496
ConnectException subclass, 500
ConnectFriendly, 497
Console class, 6, 395
  Console object, 393
  console() method, 396
  reading from console terminal, 395
  readPassword() method, 396
console terminal, reading from, 392
Console.readPassword() method, 396
constructor chaining, 382
constructors
  copy, 284
  private, 318
  superclass, 294-295
Container classes, 253
contains() method, 260, 274
containsKey() method, 274
containsValue() method, 274
Contexts and Dependency Injection (CDI), 321, 514
continuous integration (CI), 95-100
continuous refactoring, 11
continuous time, 216
controlling terminal, reading from, 392
converse method, 503
coordinates, package, 60
copy constructor, 284
copyFile() method, 397
copying files, 396-397
copyOf() method, 284
copyrights and AI, 473
CRAN (Comprehensive R Archive Network), 459, 462
createDirectories() method, 374
createDirectory() method, 372, 373, 374
createFile() method, 372, 373, 374
createNewFile() method, 413
createTempDirectory() method, 412
createTempFile() method, 412, 413
Creating Effective JavaHelp (Lewis), 30
CrossRef program, 585
CrossRefXML subclass, 586
cryptographic randomness level for numbers, 201
culture lessons, 141
CUP (parser generator), 392
CVS (Concurrent Versions System), 11
Czarnecki, David, 141

D

daemon platform threads, 422
Dahm, Markus, 586
darwinsys-api repository, 7, 8-10
Dash (Mac documentation viewer), 30
data science (DS), 455-459
data structures, 239-282
  arrays, 241-245
  Collections Framework, 246-247
  converting collections to arrays, 276
  finding objects in collections, 274-276
  generic types, 252-256
  I/O (see input/output)
  iterating over, 256-259
  List implementations, 247-252
  making your own classes iterable, 277-279
  mapping, 261-263
  multidimensional, 280-282
  Set interface to avoid duplicate Values, 259
  sorting collections, 268-273
  storing strings, 263-267
data-oriented programming (DOP), 324-326
Databricks, 457
DatagramPacket, 507, 508
DatagramSocket, 507, 508
DataInputStream class, 383, 504
DataOutputStream class, 383, 504, 520
Date class, 215, 235-237
DateFormat class, 127
dates and times, 215-237
  adding to or subtracting from dates, 231

classes, 216
converting, 225, 226-229, 235-237
difference between dates, 229-230
FormatterDates class, 119-120
formatting, 119-121, 220-225, 227
immutability of most objects in API, 216
legacy date and calendar classes, 235-237
methods, 216
recurring event calculation, 232-233
time zones and computing dates, 234-235, 378
today's date or time, 218-220
DateTimeFormatter class, 120, 144, 220-225, 227
DateTimeParseException, 227
dead string antipattern, 85
debug() method, 86
debugging
IDE tools for, 16
network-aware loggers, 82-94
testing as superior to, 77
DecimalFormat, 190
decision trees, ML classification as, 470
Deep Java Library, 471
deep learning, 469
DeepLearning4j, 471
default keyword, creating your own lambda, 337
default methods, for interfaces, 301, 305, 328
defineClass() method, 569
Deitsch, Andy, 141
delegation, as alternative to subclassing, 285
delete() method, 123
deleteCharAt() method, 123
deleteIfExists() method, 373, 376
deleteOnExit() method, 412
denial-of-service (DoS) attacks, 83
dependencies
information for Maven, 65
managing with Maven and Gradle, 59, 68-73
mock objects for unit testing, 81-82
in pom.xml file for Maven, 69
transitive, 62
dependency injection (DI), 321-323
deploy command, 64
@Deprecated annotation, 74
deprecated tag, Javadoc comment, 74
deprecation warnings, 73-74
design patterns, 283, 285
Design Patterns (Gamma, Helm, Johnson and Vlissades), 77, 283
Desktop API, 591
destroy() method, 595
DeTab subclass, 136
DI (dependency injection), 321-323
diamond operator (< >), 249
Dijkstra, Edsger, 444
directories, 359
creating and deleting, 373, 374-376
listing, 46, 360
methods for finding information on, 362-372
directory roots, 414-415
distributed filesystems, 357
documentation, Javadoc, 23-30
dollar sign ($)
newline matching, 174
R language, 466
DOP (data-oriented programming), 324-326
DoS (denial-of-service) attacks, 83
dot (.)
JAR file creating, 32
text formatting commands, 146
DOTALL flag (regex), 172
double colon (::), method reference, 341, 348
Double.equals() method, 188
Double.isNaN() method, 185
down call, 603
Downing, Troy, 586
duplicate Values, avoiding with Set, 259
Duration class, 228-229
dynamic languages, 587
dynamic loading, 551, 552-554

E

Echo server, 503, 519, 523
Eclipse IDE, 10, 13-18, 63, 79
Eclipse Jakarta EE, xix
Eclipse MicroProfile, 535
Eclipse Software Foundation, 13, 514
Effective Java (Bloch), 284, 308
Efficient Java Matrix Library (EJML), 202
EJB Timer Service, 451
elementAt() method, 250
ELKI (data mining toolkit), 471
empty files, creating, 373
encoding a text file, 401

encryption of data, JSSE, 529-532
end-of-line characters
  \r\n, 403, 520
endClass() method, 586
energy consumption issue for AI, 475
EnTab subclass, 133-136
entrySet() method, 262
Enum class, 308, 329
enum constants, 308
enum keyword, 578
enumerations, 307-311, 318
EnumMap interface, 240
EnumSet interface, 240
environment variables, 22, 42
epoch seconds, 225
epsilon, floating-point number equality, 187
equals operator (==), 288
equals() method
  array comparison, 244
  controlling case, 138
  and NaN, 187
  object formatting, 286, 288-291
  Set interface, 260
  sorting collections, 270
  tolerance, floating-point number equality, 187
equalsIgnoreCase() method, 138
erasure, 252
error handling
  floating-point number operations, 184
  network client, 500-501
  UDP packets, 507
  unit testing, 77-81
error() method, 86
escape character (\), 140, 153
Essential JNI: Java Native Interface (Gordon), 611
Evans, Benjamin, 585
events, recurring, 232-233
Exception class, 320-321
Exchange class, 528
exec() method, 588-591, 594-596
ExecAndPrint class, 594-596
execute() method
  ExecutorService, 426
  Future, 427
Executors, 422
Executors.newVirtualThreadPerTaskExecutor() method, 525
ExecutorService class, 425-426, 525, 571
ExecutorServices, 422, 447-450
exitValue() method, Process class, 595
expanding tabs, 133-137
export commands, for public API packages, 57
eXtreme Programming (Beck), 77
Extreme Programming (XP), 11, 77

F

factory methods, 190
FastR, 463
fatal() method, 86
FFI/FFM (Foreign Function and Memory API), 601-605
Field descriptor, 567
fields, accessing in reflection package, 557-561, 567-568
File class, 45, 47, 369, 412
File Watcher Service, 415
FileAttribute interface, 372
FileInputStream constructor, 381, 402
FileIO.copyFile() method, 397
filename separator (/), 46-47
FileOutputStream constructor, 373, 402
FileReader constructor, 381
files, 357
  attributes, assigning, 376-379
  change notifications, 415
  constructing or modifying Class, 573-577
  copying, 396-397
  creating and deleting, 373-374
  finding with filesystem tree, 417-419
  methods for finding information on, 362-372
  naming, 376-379
  neutral reading of, 409-411
  paths, 360-362
  reading text files, 381-384
  scanning, 384-392
  transient/temporary, 412-414
Files class, 359
  and directories, 368
  I/O options, 370-372
  informational methods, 363-364, 366-369
  legacy compatibility with File for Path, 369
  methods of, 362-372
  operational methods, 364-366
  and Path interface, 360
  temporary/transient files, 412-414

Files.copy() methods, 396
Files.createDirectories() method, 374
Files.createDirectory() method, 373, 374
Files.createFile() method, 373
Files.delete() method, 373, 374, 376
Files.deleteIfExists() method, 373, 376
Files.lines() method, 342
Files.move() method, 377-378
Files.newBufferedReader() method, 381
Files.newBufferedWriter() method, 381
Files.newDirectoryStream() methods, 369
Files.newInputStream() method, 381
Files.newOutputStream() method, 381
Files.readAllBytes method, 381
Files.readAllLines() method, 381
Files.readString() method, 383
Files::lines() method, 381
Filesystem class, 360
FileSystems.getDefault().getRootDirectories() method, 414
FileVisitor() interface, 418-419
FileWriter constructor, 373
Filter class, 529
filter() method, 369
finalize() method, 284
find() method
  Files, 369
  Matcher, 159
Find: Walking a File Tree program, 417-419
findFirst() method, 58
finisher function (Gatherer), 345
finisher() function (Collector), 340
Finnegan, Ken, 323
Firewalls and Internet Security (Cheswick), 538
first() method, 273
firstInMonth() method, 233
flatMap() method, 342
flex parser generator, 391
Float.isNaN() method, 185
floating-point numbers, 178
  checking in strings for, 180
  multiplying integer by fraction, 183-184
  and R, 464
  working with, 184-189
Flow class, 354
fluent API, 122, 218
fluent programming, 338
flush() method, 520
Flutter, 13
fold() method, 343
folds, 339
for each loop, 124, 195, 258, 277
for loop
  HashMap, 262
  iterating over structured data, 258
  one String character at a time, 124
  range of integers, 195
Ford, Neal, 330, 535
forEach() method
  HashMap, 262
  Iterable, 257, 258, 279
  Stream, 257, 258
Foreign Function and Memory API (FFI/FFM), 601-605
Fork/Join framework, 447-450
format() method
  date/time, 220
  strings, 116
FormatStyle enum, 221
formatted instance method, 116
Formatter class, 115-119, 189-193, 359
FormatterDates class, 119-120
formatting
  dates and times, 119-121, 220-225, 227
  messages, 85, 116, 198
  numbers, 189-193, 197-198
  objects, 286-293
  strings, 115-121
  text formatter sample, 146
Fowler, Martin, 96
Freeman, Eric, 283
Friedl, Jeffrey, 149
function reference, 350
functional interfaces, 328
  iterating with, 257
  with mixins, 353
  predefined lambda interfaces, 329, 335-336
  replacing inner classes with lambdas, 330
functional programming (FP), 327-356
  closures instead of inner classes, 330-333
  creating your own lambdas, 336-337
  lambdas instead of inner classes, 330-333
  method references, 348-352
  mixins, 352-353
  parallel streams and collections, 346-347
  predefined lambda interfaces, 335-336
  reactive streams and flow, 354-356
  streams, 338-347

Functional Thinking (Ford), 330
@FunctionalInterface annotation, 337
FunctionDescriptor, 603
functions
  callbacks and closures, 331
  as data, 328, 330
  versus methods, 328
  pure, 328
Future interface, 422, 427-429

G

Gamma, Erich, 77, 283
Gandhi, Raju, 535
garbage collection (GC), 103, 284
Gatherer interface, 344-346
Gatherer.of() method, 346
Gatherers class, 343
Gaussian (normal) distribution, 461
generative AI, 470, 473
generic types, 248, 252-256
Gentleman, Robert, 462
get() method
  Future, 427
  List, 250
  Map, 252
  Set, 260
getAllByName() method, 499
getByName() method, 498
getClass() method, 409, 556
getClassLoader() method, 409
getConstructors() method, 558
getEntry() method, 405
getenv() method, 42
getFields() method, 555, 558
getHostAddress() method, 498
getHostName() method, 498
getInetAddress() method, 499
getInputStream() method
  multiple network clients, 523
  Process, 592
  reading binary data, 504
  reading textual data, 501
  writing string or binary data, 518
  ZipFile, 404
getInstance() method, 319
getLastModified() method, 378
getLength() method, 508
getLocalHost() method, 499
getMethod() method, 559-561
getMethods() method, 555, 558
getName() method, 404
getNestHost() method, 567
getNestMembers() method, 567
getOpenAPIKey() method, 475
getopt library, 176
getOutputStream() method
  multiple network clients, 523
  reading binary data, 504
  reading textual data, 501
  writing string or binary data, 518
getResource() method, 409-411
getResourceAsStream() method, 409-411
getServerSocket() method, 530
getYear() method, 466
git clone command, 7
git pull command, 7
Git SCM, 11
GitHub, downloading code examples, 7, 12
global arena, 603
global constants (FP), 328
Goetz, Brian, 326, 454
Google Flutter, 13
Google Guice, 321
Gordon, Rob, 611
Gough, James, 585
GPUs, 472
GraalVM, 39-41, 463, 599-601
Gradle
  automating with, 66-68
  and CLASSPATH, 23
  dependency management, 71-73
  JUnit with, 79
  and Maven, 68
  WAR file creation, 38
gradle dependencies command, 71
Grails, 587
grammatical structures, input/output with, 391-392
Grand, Mark, 586
green threads, 429
Gregorian calendar, 228
grep utility, 148, 175
Groovy, 60, 66, 587
group() method, 160, 163, 168
groupCount() method, 162
grouping to specify part of regex, 160
gu command line tool, 601
Gupta, Arun, 514

gzip compression, 408

H

Hacking Exposed (McClure), 538
Hadoop, 455
hallucinations, in AIs, 474
Hamcrest matchers, 79
Handler class, 523
hardcoded mock object, 81
Harold, Elliotte Rusty, 194, 381, 494, 538
hashCode() method, 288-292
HashMap, 240, 261-263, 265
HashSet, 240
Hashtable, 240, 263, 265
hasMoreElements() method, 123
hasMoreTokens() method, 113
hasNext() method, 277
Head First Design Patterns (Freeman and Robson), 283
Head First Software Architecture (Gandhi, Richards, Ford), 535
headMap() method, 273
headSet() method, 273
heap memory, 603
hexadecimal numbers, 195
Hitchens, Ron, 380
HTTP protocol, 526-529
HTTP/REST web client, 494-496
HttpClient class, 494-495, 496
HttpHandler class, 529
HttpRequest object, 495
HttpResponse object, 495
HttpURLConnection class, 478, 494, 495
human time, classes for, 217
human-readable number formatting, 192

I

I18N resources, 141-143, 145
IBM J9, 3
IDE (see Integrated Development Environment)
IDE warning messages, 75-77
Ihaka, Ross, 462
Image class, 487
image poisoners, generative AI, 473
ImageIO, 484
ImageModel, 484-485
images
  classification of, 469
  generating, 484-485
  inferences from, 486-487
immutable data, 324, 329
implicitly declared class, 4-7
import statement, 51
incremental compiling features, IDE, 13
indent() method, 128
indent(int n) method, 128
indenting strings, 128
indexOf() method, 113, 274
Inet6Address, 499
InetAddress object, 498-499, 516
InetAddress.getByName() method, 517
inferences from images, 486-487
inferencing, in machine learning, 469
info() method, 86
initializer function, 344
initialValue() method, 440
Inline::Java (Perl module for Java), 588
inner classes, 295-296, 330-333, 565-567
input/output (I/O), 357-419
  attribute changes to files, 376-379
  binary data, 379-381, 383
  console/controlling terminal, reading from, 392-396
  converting character sets, 401
  copying files, 396-397
  creating and deleting files or directories, 373-376
  directory roots, 414-415
  duplicating streams, 398-401
  end-of-line characters, 402
  File Watcher Service, 415
  JAR archives, 404-408
  methods for file and directory information, 362-372
  name changes to files, 376-379
  network text data transfer, 501-503
  neutral reading of files, 409-411
  Path object, 360-362
  platform-independent, 403-404
  reading text files, 381-384
  reassigning standard streams, 397
  scanning files, 384-392
  standard input, reading from, 392-395
  streams, 379-380
  terminal, reading from, 392-396
  text, reading, 379-381
  transient/temporary files, 411-414

walking a file tree, 417-419
Zip archives, 404-408
InputStream class, 359
and ClassLoader, 409
closing to prevent memory leaks, 383
creating URL, 510
reading binary data, 379-380
reassigning standard streams, 398
with sockets, 502, 518-520
InputStreamReader class, 393, 394, 401
insert() method, 123
instance main, 4-7
instance() method, 318
Instant class, 228
Integer class, 194, 561
Integer.parseInt() method, 194, 395
Integer.TYPE, 561
integers, 178
checking in strings for, 180
converting to/from bit-series numbers, 194-195
multiplication by fraction, 183-184
rounding, floating-point numbers, 188
very large numbers, 210-212
working on range of, 195
Integrated Development Environment (IDE), 1-2
batch refactoring for, 75-77
compiling, running and testing with, 11, 12-18
Maven support for, 63
integrator function, 344
IntelliJ IDEA, 13-18
inner classes as lambdas, displaying, 333
JUnit with, 79
Preview, enabling, 5
TeamCity, 96
@interface annotation, 577, 578
interfaces, 300
(see also functional interfaces)
callbacks via, 300-303
with default, static, and private methods, 301, 305-307, 328
finding server-side network, 517
predefined lambda, 335-336
sealed, 316
internal iteration, 258
internationalization, 141
choosing particular locale, 143-145
controlling case, 138
dates and times, 222
formatting strings for, 117
with I18N resources, 141-143, 145
interned box values, 183
Internet Protocol (IP), 499
interpolation, string, 121
introspection, JUnit's use of, 78
IntStream class, 195
IntStream.rangeClosed() method, 257
IntUnaryOperator class, 337
invoke() method, 559, 562
invokeExact() method, 562
IOException, 359, 382, 394, 523
IP (Internet Protocol), 499
IPv6, 499
isCancelled() method, 427
isDone() method, 427
isEmpty() method, 314
isNan() method, 185
isNestMateOf() method, 567
isPresent() method, 314
isProbablyPrime() method, 212
iterable data, publishing your structure as, 277-279
Iterable interface, 277-279, 306
Iterable.forEach() method, 257, 258
iterating over structured data, 258
Iterator interface, 114, 263, 277-279
iterator() method, 277, 279

J

Jackson API, 476, 542-544, 578
jackson.org parser, 540
Jakarta EE Cookbook (Moraes), xix
Jakarta Enterprise API, 541
Jakarta Enterprise Edition (Jakarta EE), xix
HTTP protocol serving, 526
server-side network, 514
WAR file creation, 38
Jakarta Persistence API, 578
jakarta.json (jsonp), 541
JAR (Java ARchive) files
and CLASSPATH, 21
creating with Maven, 62
input/output, 404-408
multi-release for Java versions, 35-37
packaging and running, 32-35
jar tool, 32, 38

Java
books on, xix
build tools for, 96, 588, 592
calling from native code, 605-611
calling within R session, 465-466
compatible versions, xiii
compiling and running, 1-47
conceptual diagram, xiv
data science infrastructure role, 455
downloading, 1
FFI/FFM for non-Java code access, 601-605
javax.script to call other languages, 596-599
machine learning packages, 471
networking in, 492-493
packages, 49-51
running external programs from, 587-611
running program to capture output, 592-596
Java 16, 617-619
Java 17, 619
Java 18, 620
Java 19, 620
Java 20, 621
Java 21, 622-623
Java 22, 623-624
Java 23, 624
Java 24, 625-626
Java 8 Lambdas (Warburton), 330
Java 9 Modularity (Mak and Bakker), 52
java command, 2
Java Concurrency in Practice (Goetz), 454
Java Development Kit (JDK), 1-2
Java EE (see Jakarta Enterprise Edition)
Java EE 7 Essentials: Enterprise Developer Handbook (Gupta), 514
Java Enhancement Proposal (JEP), 617
Java Enterprise Edition (Java EE), xix
Java Flight Recorder, 101
Java Generics and Collections (Naftalin and Wadler), 252
Java I/O (Harold), 194, 381
Java Internationalization (Deitsch and Czarnecki), 141
Java Language Reference (Grand), 586
Java Message Service (JMS), 538
Java Message Service (Richards), 538
Java Micro Edition (Java ME), xix
Java Mission Control, 101
Java Module System, 49
Java Native Interface (JNI), 605-611
Java Network Programming (Harold), 494, 538
Java NIO (Hitchens), 380
java package, 50
Java Performance, 585
Java Persistence API, 57
Java Platform Module System (JPMS), 52-59
Java Secure Socket Extension (JSSE), 512, 529-532
Java Servlet and JSP Cookbook (Perry), xix
Java Software Development Kit (SDK), 1
Java Specification Request (JSR), 215
Java Util Logging (JUL), 83, 91-94
Java Virtual Machine (Downing and Meyer), 586
Java Virtual Machine (JVM), 551
java.awt.Desktop class, 45, 591
java.awt.TaskBar class, 45
java.base module, 56
java.desktop module, 56
java.io package, 109
java.io.File class, 47, 359
java.io.IO class, 6
java.lang.Character, 152
java.lang.CharSequence interface, 159
java.lang.Class, 551
java.lang.classfile, 573
java.lang.Comparable, 270
java.lang.Enum, 308
java.lang.foreign, 603
java.lang.Integer, 194
java.lang.Math, 179
java.lang.Math.random() method, 199
java.lang.Object, 251, 284, 422, 556
java.lang.object
and equals() method, 289
and toString() method, 287
java.lang.Override, 578
java.lang.Package, 563
java.lang.reflect, 551, 555, 557-561
java.lang.Runtime, 588
java.lang.String, 149, 159, 558
java.math, 179, 210
java.net api, 476
java.net.Socket, 496
java.net.URLClassLoader, 569
java.nio package, 149, 170
java.nio.CharBuffer, 159
java.nio.file.Files, 359, 362-372, 373, 412

java.nio.file.Path, 360
java.nio.file.WatchService, 415
java.se module, 56
java.security.SecureRandom, 201
java.text package, 190
java.text.CollationKey, 270
java.text.Format, 116
java.time package, 217, 270
java.time.format.DateTimeFormatter, 220
java.time.LocalDate.now() method, 465
java.util package, 240
    (see also data structures)
java.util.Arrays, 240
java.util.BitSet, 195
java.util.concurrent, 422, 454
java.util.Concurrent, 444
java.util.concurrent.Flow, 354
java.util.concurrent.locks, 436-437
java.util.Date, 73-74
java.util.Enumeration, 259
java.util.Formatter, 115-119, 384
java.util.function, 335
java.util.function.Consumer, 257
java.util.Iterator, 259
java.util.List, 540
java.util.Locale, 190
java.util.Map, 540
java.util.Observable, 293
java.util.Optional, 314
java.util.prefs.Preferences, 264
java.util.Properties, 43, 146, 264
java.util.Random, 199-201
java.util.Random class, 461
java.util.regex, 156-160
java.util.Stack, 212
java.util.stream.Stream, 339
java.util.Timer, 451-452
java.util.zip, 585
java.util.zip.ZipFile, 404-408
javac command, 2, 31, 298
JavaCC (parser generator), 391
JavaCompiler, 570-573
Javadoc (javadoc tool), 23-30, 285
JavaFileManager interface, 573
JavaHelp, 30
JavaMail API, 57
javap class printer, 6, 298, 554
javapvr command alias, 4
JavaServer Faces (JSF), 514
JavaServer Pages (JSP), 514
javasrc repository, 7, 8
javaw command, 2
javax package, 50
javax.annotations.PostCreate, 582
javax.json package, 548
javax.persistence package, 578
javax.script package, 587, 596-599
javax.tools.JavaCompiler, 570, 573
JAX-RS, 532-535
JBoss Weld CDI for Java Platform (Finnegan), 323
jcall() function, 466
jdeps command, 104
JDK (Java Development Kit), 22, 104-105
jdk.unsupported module, 54
Jenkins (continuous integration), 96
JEP (Java Enhancement Proposal), 617
Jersey, 534
JetBrains Toolbox App, 13
JFlex, 392
JGrep program, 175
jlink command, 1, 104-105, 107
JMS (Java Message Service), 538
JNI (Java Native Interface), 605-611
Joda-Time package, 215
join() method
    StructuredTaskScope, 443
    threading, 431-432
jpackage command, 35, 106-107
JParsec, 391
JPMS (Java Platform Module System), 52-59
JRuby, 587
JSF (JavaServer Faces), 514
JShell, xv, 19-21, 460
JSON (JavaScript Object Notation), 539-550
    formatting and parsing with Jackson, 476-479
    generating directly, 541-542
    read/write with Jackson, 542-544
    read/write with JSON-B, 546-547
    read/write with org.json API, 544-546
    structure of, 540-541
JSON arrays, 540
JSON objects, 540
JSON Pointer, 548-550
JSON-B, 546-547
JSON-Java (org.json), 544-546
JSONArray, 546

JSONObject, 546
JSP (JavaServer Pages), 514
JSR (Java Specification Request), 215
    (see also dates and times)
JSSE (Java Secure Socket Extension), 512, 529-532
JUL (Java Util Logging), 83, 91-94
Julian calendar, 228
JUnit testing, 57, 78-80
JVM (Java Virtual Machine), 551
JVM shutdown hook, 413
jwebserver command, 526, 528
Jython, 587

K

Kaffe package, 554
Kapeli, 30
Kernighan, Brian W., 136
key/value collections, 240
    (see also data structures)
Kotlin, 66

L

lambda expressions, 195
    creating your own lambdas, 336-337
    and functional interfaces, 329, 335-336
    instead of inner classes, 330-333
    with network-based loggers, 90, 94
lanes (Vector), 204
LangChain4j, 472, 479-487
    AI Service interface, 480-482
    ChatGPT, 479
    ChatMemory to retain conversation, 483
    generating images, 484-485
    inferences from images, 486-487
    ollama package, 488-490
language lessons, 141
large language models (LLMs), 470
Large Language Models (LLMs), 479, 488-490
large numbers, handling, 210-212
last() method, 273
layout() method, 461
lazy evaluation/lazy loading, 319
Lea, Doug, 422, 454
legacy dates, 235-237
Level class, 87
Lewis, Kevin, 30
lex parser generator, 391
lifecycle phases, Maven and Gradle, 60
lines() method
    BufferedReader, 395
    Files, 342
    Stream<String> (String method), 129
linked lists, 239
    (see also data structures)
LinkedList class, 248, 251
Linker class, 603
List interface, 195, 246, 247-252
    ArrayList class, 248-249
    ArrayList(), 284
    conversion from array to List, 250
    data structures, 247-252
    and end-of-line characters, 404
    LinkedList class, 251
    platform-independent code and end-of-line character, 404
    reversed() method, 133
    Vector class, 250
list() method, 368
List.of() method, 250
List< String> Files.readAllLines() method, 381
ListIterator interface, 252
ListMethods class, 558
LLMs (large language models), 470
LLMs (Large Language Models), 479, 488-490
load() method, 45, 58
LoadAverage class, 53
loadAverage() method, 53
loadClass() method, 569
LocalDate class, 218
LocalDate.now() method, 465, 466
LocalDateTime class, 218, 230, 505
Locale class, 190
Locale constructor, 144-145
Locale.getAvailableLocales() method, 143
Locale.setDefault(newLocale) method, 144-145
locales for internationalization, 141-145
localization (see internationalization)
LocalTime class, 218, 230
Lock interface, 436-437
lock() method, 436
lockInterruptibly() method, 436
locks, 422, 437-440
log() method, 86
Log4j, 82
log4j2.properties file, 87
Logger class, 86
LoggerFactory class, 84

LoggerFactory.getLogger() method, 84
logging, 82-94
LogManager.getLogger() method, 87
Lombok, 297, 299
LongStream class, 195
lookingAt() method (Matcher), 159
Loukides, Mike, 149, 455
Luma Labs, 473

M

machine learning (ML), 469-490
  ChatGPT, 470, 476-480
  ethical considerations in AI, 473-475
  Java packages for, 471
  LangChain4j
    AI Service interface, 480-482
    ChatGPT, 479-480
    ChatMemory to retain conversation, 483
    generating images, 484-485
    inferences from images, 486-487
    LLM, running locally, 488-490
main() method, 6
  instance main, 4
  threading, 422
Main-Class, 32
Mak, Sander, 52, 342
make tool, 98
Mallet (ML for text), 471
manifest, JAR file, 33
Map data structure, 246
Map implementation, 261-263, 286, 288
Map interface, 252
map types, 240
Map.Entry, 262
Map.get() method, 252
Map.put() method, 252
mapConcurrent() method, 343
mapping operation, streams, 338
MapReduce, 329
Markdown Javadoc, 28-29
Mastering Regular Expressions (Friedl), 149
match() method, 159, 168
Matcher API, 160
Matcher class, 157-171
matches() method, 156
matching text (regex), finding, 162-165
Math class, 179, 199
Math.round() method, 188
matrices, multiplying, 201-203
Matrix class, 203
Maven (see Apache Maven)
Maven Central, 65-66, 69
McClure, Stuart, 538
ME (Java Micro Edition), xix
memory management, JNI issue, 611
MemorySegment, 603
MessageFormat class, 85, 116, 198
MessageWindowChatMemory (LangChain4j), 483
metacharacters (regex), 150-152
metadata facility, 30-31
metadata, using annotations to provide, 578
Method descriptor, 567
method references, 348-352
MethodHandle API, 562-563, 603
methods
  abstract, 304-305
  acceptor-type, 335
  accessing in reflection package, 557-561, 567-568
  anonymous, 330
  boolean, 363-364
  dates and times, 216
  defaults for interfaces, 301
  factory, 190
  file and directory search, 362-372
  Files class
    informational methods, 363-364, 366-369
    operational methods, 364-366
  versus functions, 328
  for interfaces, 301, 305-307, 328
  measuring performance, 101-102
  Optional to pass values into, 315
  stream type, 338
MethodType object, 562
Meyer, Jon, 586
microservice architecture (MSA), 535
Microsoft VSCode, 13, 17
migration from one API to another, 75-77
mini-JDK distribution, creating, 104-105
minus() method, 231
mixins, 337, 352-353
ML (see machine learning)
mock objects, 81-82
Mockito framework, 80, 81-82
Model-View-Controller (MVC), 323
Moderne, 77

Modifiers object, 555
module-info.java file, 52-59
modules, and packages, 49-51
Moraes, Elder, xix
MoreUnit plug-in, 79
move() method, 377-378
MSA (microservice architecture), 535
Multicast, 508
multidimensional structures, 280-282
MULTILINE flag, 172, 173, 174
multiline strings, 110, 128, 404
multiple-client handling with threading, 522-525
multiprocessing versus multi-tasking, 421
multithreaded applications (see threading)
MVC (Model-View-Controller), 323
mvn (see Apache Maven)

N

n indicator, Gradle, 72
Naftalin, Maurice, 252
name and version, assigning, 60
named groups (regex), 165
namespace, class loader, 569
NaN (Not a Number), 185, 188
NAT (Network Address Translation), 499
native code, 587, 605-611
native keyword, 606
native-image tool, 40
ND4J package, 202
negate() method, 210
NEGATIVE_INFINITY constant, 184
Nelson, Mark, 535
Neon, 204
nest-based API, 565
nested members of a class, accessing, 565-567
NetBeans, 13-18
netlog API, 83
Network Address Translation (NAT), 499
network router, 516
network-aware loggers, 82-83
networking (see client-side network; server-side network)
NetworkInterface class, 517
neural nets, 469
new operator, 603
newInstance() method, 552, 553
newline matching in text, 173-175
next() method, 277
nextBoolean() method, 200, 461
nextBytes() method, 200
nextDouble() method, 200, 461
nextFloat() method, 200
nextGaussian() method, 200, 461
nextInt() method, 199, 200
nextLong() method, 200
nextRandom() method, 461
nextToken() method, 113
Nexus Repository Manager, 65
NIO package, 380, 525
no-echo password reading, 396
non-daemon platform threads, 422
nonfield computations before calling superclass, 294-295
nonprintable characters, adding to string, 140-141
nonpublic class, 295-296
normal (Gaussian) distribution, 461
NoRouteToHostException subclass, 500
NoSuchFileException, 374
notifyAll() method (Thread), 444
now() method, 218
null device, 47
null references, 314, 411
NullPointerException (NPE), avoiding, 313-315
numbered groups (regex), 165
NumberFormat class, 127, 144, 189-193
numbers, 177-213
  arithmetic operations on large numbers, 203-207
  complex, 207-209
  converting integers to bit-series numbers, 194-195
  converting to/from objects, 182-183
  floating-point (see floating-point numbers)
  formatting, 189-193, 197-198
  generating pseudorandom, 179, 199-201
  integer multiplication by fraction, 183-184
  integers (see integers)
  matrices, multiplying, 201-203
  TempConverter program, 212-214
  validating numbers in strings, 180-181
  Vector numeric types, 204
  very large numbers, 210-212
Numerical Recipes book series (Press et al.), 203
NumFormatDemo program, 191

0

Object references, 560
object references, arrays, 241
object serialization, 506
object stacks, 132-133, 252-255
Object types
  array comparison, 245
  ArrayList storage for, 248
object wrapper classes, numeric types, 178
object-oriented programming (OOP), 52, 283-326
  callbacks via interfaces, 300-303
  constructor simplification, 294-295
  custom exception classes, 320-321
  data-oriented programming, 324-326
  dependency injection, 321-323
  design patterns, 285
  enumerations, 307-311
  formatting objects, 286-293
  generality versus specificity, 284
  improving interfaces with methods, 305-307
  inner classes, 295-296
  Java API, use cases, 284
  Javadoc's importance, 285
  NullPointerException, 313-315
  polymorphism and abstract methods, 304-305
  simplifying data objects, 297-299
  Singleton pattern enforcement for JVM, 317-320
  subclassing, 285, 315-317
  type pattern matching, 311-313
  unit testing in, 77
ObjectInputStream, 504, 506
ObjectOutputStream, 504, 506
objects
  arrays of, 241
  converting to/from numbers, 182-183
  finding in collections, 274-276
  formatting, 286-293
  as functions in FP, 328
  mock objects, 81-82
octal numbers, 194
ofDays() method, 231
off-heap memory, 603
ofLocalizedDate() method, 221
ofLocalizedDateTime() methods, 221
ofLocalizedTime() method, 221
ofPattern() method, 222
ollama package, 488-490
OpenAI, 475
OpenBSD, 493
OpenJDK, 2, 3
OpenOption values, 370
OpenRewrite, 75-77
operating systems
  console/controlling terminal, reading from, 395
  default locale for internationalization, 142
  epoch seconds, 225
  filesystem organization of multiple drives or partitions, 414
  JNI considerations, 606, 610
  multiprocessing support, 422
  networking issue, 492
  ollama package, 488
  permissions for deleting files, 375
  platform-independent code, 403-404
  platform-specific installers, creating, 106-107
  and running Java in other languages, 587
  runtime environment information, 42
  standard input and output, 392-395
  stored preferences, 264
  threading, 429
  Unicode character handling, 131
  Windows versus Unix canonical names, 368
  writing code that adapts to, 45-47
Optimizing Cloud Native Java (Evans and Gough), 585
Optional methods, 278, 314
Optional wrapper class, 314-315
Optional.empty() method, 314
Optional.of() method, 314
Optional.ofNullable() method, 314
orElse method, 314
org.apache.logging.log4j package, 86
org.json API, 544-546
org.json parser, 540
OutputStream class, 359
  closing to prevent memory leaks, 383
  with sockets, 502, 518-520
  writing binary data, 379-380
OutputStreamWriter class, 401
@Override annotation, 289, 578
O'Reilly Online Learning Platform, xx
O'Reilly, Tim, 149, 470

P

package command (Maven), 64
package statement, 49, 51
packages, 49-51
  export commands for public APIs, 57
  and javadoc tool, 25
  listing classes in, 563-565
  machine learning, 471
  with Maven and Gradle, 59
  naming conventions, 51
  WAR files as, 38
packets, UDP, 507
parallelism, 342
  (see also streams)
  processing with Fork/Join, 447-450
  streams and collections, 346-347
parallelStream() method, 346
parameter order in format code, 118
parentheses() method, regex grouping, 163
parenthesized groups (regex), 160
Parlog, Nicolai, 326
Parschiv, Eugen, 472
parse() method, 226-227
parser generator, 391-392
parsers for JSON, 540
Parsing Expression Grammar (PEG), 392
passwords, reading with Console class, 396
Path class, 45, 359
  and Files class, 363
  informational methods, 366-369
  legacy compatibility with File, 369
  methods of, 360-362
  temporary files and directories methods, 412
Path.of() method, 360
paths, files and, 360-362
pathSeparator property, 47
pathSeparatorChar property, 47
Pattern class, 157-160
  controlling case, 171-172
  finding matching text, 162-165
  matching accented or composite characters, 172
  printing all occurrences of pattern, 168-171
  replacing matched text, 165-168
pattern matching, 159, 311-313, 324
  (see also regular expressions)
pattern-based date/time formatting, 222-225
Pattern.compile() method, 171-172, 174
pattern.matcher() method, 158
peek() method, 253
Peek, Jerry, 149
PEG (Parsing Expression Grammar), 392
percent symbol (%), printf formatting, 224
performance timing, 101-103
Period class (date/time), 228, 229, 231
Period.between() method, 229-230
Perl, 177, 588
permissions, reading and writing files, 375
Perry, Bruce, xix
pipelines (see streams)
piping, 398
platform threads, 422
platform-independent code, 403-404
platform-specific installers, creating, 106-107
Plauger, P. J., 136
plug-in-like classes, 581-583
plurals, formatting numbers with correct, 197-198
plus (+) operator (string concatenation), 121
plus() method, 231
poll() method, 525
polymorphism, 305
POM file, 63
pom.xml file, 60, 61-62
pop() method, 253
POSITIVE_INFINITY constant, 184
POSIX, 371
PosixFileAttributeView subtype, 372
PosixPermission class, 371
Powers, Shelley, 149
Predicate interface, 335
Preferences class, 264-265
preferences.get() method, 265
Press, William, 203
primitive types, 177, 241
print() method, 401
printf() function ©, 189, 213
printf() function, C, 116
printf() method (Java), 384
printing class information, 554-555
println() method, 6
  JSON, generating, 541
  for platform-independent code, 403
  PrintWriter, 403
  ServerSocket, 520
  text formatter, 146
  and write(), 401

PrintStream class, 384, 398
PrintStream.write() methods, 398
PrintWriter class, 384, 401, 403, 501
PrintWriter.println() method, 541
privacy issues for AI, 473
private classes, 295-296
private constructor, Singleton pattern, 318
private methods, 306, 566, 567-568
PRNG (pseudorandom number generator), 179, 199-201
process() method, 6
Process.getInputStream() method, 592
ProcessBuilder, 588, 592
ProdConsThreadAPI, 444-446
producer-consumer implementation, 444-446
profilers, 101-103
Project Loom: Fibers and Continuations, 454
Project Panama, 601
proleptic Gregorian calendar, 228
Properties class, 43-45, 265-267
Properties file, 146
properties files, locale, 143, 145
protected defineClass() method, 569
ProtectionDomain, 586
prototyping code with JShell, 20
pseudorandom number generator (PRNG), 179, 199-201
Publisher interface, 354
pure functions, 328
push() method, stack, 253
push-down stack, 253
put() method, 252
Python, 455, 587, 599-601

Q

Quartz, 451
Queue, 422
Queue interface, 246, 444-446

R

R Foundation for Statistical Computing, 462
R language, 456, 459-468
  calling Java code within R session, 465-466
  choosing an implementation, 462
  FastR, 463
  GUI frontend for, 461
  interactive use of, 459-462
  Renjin to access, 463-465
  web page display of data, 467-468
random numbers, 179, 199-201
random() method, 199
range() method, 195
rangeClosed() method, 195
Rats! (parser generator), 392
React Native, 356
Reactive API, 354
read() function, in Apache Spark, 456
read() method
  and end-of-line character issue, 403
  IOException, 394
  Reader, 384
read-evaluate-print loop (REPL) interface, 19-21, 460
Reader class, 359
  character conversion, 394
  closing to prevent memory leaks, 383
  reading text data, 379-380
readers-writer lock, 437-440
reading files (see input/output)
readInt() method, 504
readLine() method
  and end-of-line characters, 403
  for platform-independent code, 403
  standard input, reading from, 394
  strings, 110, 174
readLock() method, 437
readUnsignedByte() method, 504
readUnsignedShort() method, 504
ReadWriteLock interface, 437-440
real numbers, 178
record data type, 297-299, 313, 324, 329
recurring event calculation, 232-233
recursive descent parser, 392
RecursiveAction class, 447
RecursiveTask class, 447
REDemo, 153-155
redirection, 398
ReentrantReadWriteLock class, 437
refactoring, IDE feature, 11, 13, 15, 75-77
reflection, 551-586
  annotations, 31, 577-583
  Class descriptor, 556-557
  class file construction or modification, 573-577
  ClassLoader class, 568-570
  CrossRef program, 585
  dynamic loading/instantiating of classes, 552-554

fields, accessing, 557-561
JavaCompiler, 570-573
listing classes in a package, 563-565
methods, accessing, 46, 557-561
nested members of a class, 565-567
opens command to export to, 57
printing class information, 554-555
private fields/methods, accessing, 567-568
timing program, 583-585
and Unsafe object, 53
regular expressions (regex), 147-176
controlling case in, 171-172
finding matching text, 162-165
grouping to specify part of, 160
JGrep program, 175
matching accented or composite characters, 172
matching String to regex, 156-160
newline matching in text, 173-175
printing all occurrences of pattern, 168-171
regex package, 150-152
regex public API, 157-159
replacing matched text, 165-168
splitting strings, 114
syntax of, 149-155
very large numbers, 212
reliability issue for AI, 473
remote interface, 300
Remote Methods Invocation (RMI), 538
remove() method, 259, 277, 279
Renjin, 463, 588
REPL (read-evaluate-print loop) interface, 19-21, 460
replace() method, 123
replaceAll() method, 166, 167
replaceFirst() method, 166, 167
replacing matched text, 165-168
request() method, 354
ResourceBundle class, 141
ResourceBundle.getBundle() method, 142, 145
REST API, 476
REST-based web services, 494
RESTEasy, 534
RESTful Java with JAX-RS 2.0, 2nd Edition (Burke), 496
RESTful server with JAX-RS, 532-535
retainAll() method, Set, 261
@Retention annotation, 581
reverse() method, 123, 132
reversed() method, 133
reversing strings, 132-133
Richards, Mark, 535, 538
Ritchie, Dennis, 225
rJava library, 465-466
RMI (Remote Methods Invocation), 538
Robson, Elisabeth, 283
Roman numeral formatting, 193
RomanNumberFormat class, 193
round() method, 188, 189
rounding error, 178
rounding, floating-point numbers, 188
Ruby, 587
run() (method of Runnable interface), 423, 453
scheduling tasks, 451, 452-454
threading, 424, 430, 523
running an application, IDE tools, 12-18
Runtime class, 41
Runtime.exec() method, 587
RuntimeException class, 320-321
RxDart, 356
RxJava, 356

S

S language, 462
saveFile() method, 453
Scala, 328, 455, 587
Scalable Vector Extensions (SVE), 204
scan() method (Gatherer), 343
scanf() function, C, 388
Scanner class, 359, 387-391, 395
scanning files, 384-392
scheduling tasks with threading, 451-454
SCM (source code management) system, 11
scope values, Maven dependency management, 70
ScopedValue class, 442-444
ScriptEngine interface, 463
ScriptEnginesList program, 596
Scripting Engine framework, 464, 596
scripting languages, 587
SDKMAN, 2
sealed types, 315-317, 324
search() method
Collection, 274
Predicate, 336
security
avoiding user-provided format strings, 117
Jenkins CI server, 96

network-based loggers, 83
SecurityManager (deprecated), 586
select() method, 525
Semaphore class, 444
separation of concerns, and DOP, 324
separator property, 47
separatorChar property, 47
sequential versus threaded processing, 421
serialized Java object data, network transfer of, 504, 506
server-side network, 513-538
client contact with socket server, 496-497
finding network interfaces, 517
HTTP protocol, 526-529
Java Util Logging, 83, 91-94
layering of, 491
multiple-client handling with threading, 522-525
RESTful server with JAX-RS, 532-535
returning a response (string or binary), 518-522
securing web server with TLS, 529-532
securing with TLS, 529-532
SLF4J, 83, 84-86
Unix domain sockets, 535-538
writing socket-based server, 514-517
ServerSocket class, 514-517, 526, 530
service end of network connection, 514
service feature, JPMS, 57
servlet, 514
Servlet API, 38
Set data structure, 246
Set interface to avoid duplicate Values, 259
set methods, number formatting, 190
Set<PosixFilesPermission>, 372
setAccessible() method, 567
setClock() method, 219
setLastModified() method, 378
setLastModifiedTime() method, 377
setMinimumIntegerDigits() method, 190
setOut() method, 398
setPort() method, 508
setReadable() boolean method, 378
shape (Vector), 204
Shape class, 301
shared arena, 603
sharing data among threads, 440-444
shoulder surfing, defending against, 396
shutdown() method (ScopedValue), 443
shutDown() method (Thread), 424, 426
Simple Logging Facade for Java (SLF4J), 83, 84-86
SimpleFileServer, 526
Single Instruction Multiple Data (SIMD), 204
Singleton pattern enforcement for JVM, 317-320
size() method, 276
slash (/)
filename separator, 46-47
JSON Pointers, 548
SLF4J (Simple Logging Facade For Java), 83, 84-86
snippets, Javadoc code, 26-28
SOAP-based web services, 494
sock.receive() method, 508
sock.send() method, 508
sockaddr_in structure, 496
Socket class, 496, 499, 502
socket code, 491
Socket object, 515
socket server, contacting, 496-497
SocketException class, 500
SocketFactory, 530
sockets (see client-side network; server-side network)
Software Tools (Kernighan and Plauger), 136
sort() method, 268-270
SortedSet interface, 261
sorting collections, 268-273
source code management (SCM) system, 11
source-code launcher, 4
spaces to tabs conversion, strings, 133-137
Spark, 455, 456-459
species of vectorization, 204
split() method, 114, 342
Spliterator, 330
Spring AI, 471, 472, 490
Spring DI, 321-323
Spring Framework, 578
square brackets ([ ]), Markdown, 28
SSE (Streaming SIMD Extensions), 204
SSL certificate management, 530-532
SSLServerSocketFactory class, 530
Stack class, 132, 253-256
standard input, reading from, 392-395
Standard JDK, compiling and running with, 2-3
start() method
Matcher, 162

ProcessBuilder, 588
Thread, 424
startClass() method, 586
startsWith() method, 154
startWalkingAt() method, 418
static accessor method, 318
static import feature, 352
static methods, improving interfaces, 306
static String.format() routine, 116
stderr stream, 459
stdout stream, 459
Stevens, Richard, 512
stop() method, 424
Stream class, 329, 338-339
Stream Gatherer, 343-346
Stream interface, 359
stream() method, 346
stream-based TCP connection, 507
Stream.flatMap() method, 342
Stream.forEach() method, 257, 258
Stream.sorted() method, 368
Stream<Path> list() method, 368
Stream<String> lines() method, 129
Streaming SIMD Extensions (SSE), 204
streams, 343
bytes to be read or written, 359
Collector terminal function, 339-342
duplicating, 398-401
in functional programming, 338-347
input/output, 379-380
parallel, 346-347
reassigning standard, 397
simplifying processing, 338-339, 343-346
Streams API, 343
StreamTokenizer class, 212
String class, 109
controlling case, 138-139
getting methods list, 558
immutability of, 329
Java support for, 149
methods for alternative number base conversions, 195
pattern matching, 159
sorting collections, 268
string literals, internal storage of, 110
String objects, and StringBuilder, 121
String.boolean matches() method, 156
String.class, 560
String.format() method, 110, 116, 541
string.formatted() instance, 116
String.join() method, 123
String.repeat() method, 126
String.split() method, 114, 342
String.substring() methods, 164
String.trim() method, 114
StringAlign class, 126-127
StringBuffer class, 122, 159
StringBuilder class, 121-123
aligning strings, 126
converting Unicode characters to strings, 130
multiline text strings, 111
pattern matching, 159
reversing characters in a string, 132
string concatenation, 110
StringBuilder.append() method, 103
StringBuilder.length() method, 123
strings, 109-146
aligning, 126-129
concatenating, 110, 121-123, 178, 197, 286-288
controlling case, 138-139
creating messages with I18N resources, 141-143
formatting, 115-121
immutability of, 111-112, 166
indenting and unindenting, 128
interpolation of, 121
java.time object conversion, 226-229
localization (see internationalization)
nonprintable characters, adding to string, 140-141
number conversion, 178
processing one character at a time, 124-125
regular expressions (see regular expressions)
reversing, 132-133
spaces to tabs conversion, 133-137
storing in properties and preferences, 263-267
StringBuilder, 121-123
substrings, 113
text formatter sample, 146
tokenizing, 113-115
Unicode character conversion, 129-131
validating numbers in, 180-181
StringTemplate class, 121
StringTokenizer class

breaking strings apart, 113
joining strings, 123
scanning files, 385-387
splitting string into words, 114
stringtree.org JSON parser, 540
strip() method, 114
stripIndent() method, 114, 129
stripLeading() method, 114
stripTrailing() method, 114
StructuredTaskScope object, 443
structuring data (see data structures)
subclasses, 301
and abstract methods, 304-305
creating custom exceptions, 320-321
with CrossRef, 586
in object-oriented programming, 285
with sealed types, 315-317
and stream classes, 399
subMap() method, 273
SubmissionPublisher class, 354
submit() method, 426
Subscriber interface, 354
subSet() method, 273
substring() method, 113
aligning, 126
breaking strings apart, 113
finding regex matching strings, 168
MethodHandle API, 562
substrings, 113
Subversion, 11
Suleyman, Mustafa, 475
super() method, 294
superclass constructor, 294-295
supplier() function, 340
Swing, 46
synchronization
methods with StringBuffer, 122
threads, 432-437
synchronized keyword, 432-436
synchronizers, 422
synchronous sockets, 492
System class, 41, 393, 398
system-dependent code, 44
(see also operating systems)
System.console() method, 393, 396
System.currentTimeMillis() method, 226, 584
System.getenv() method, 42
System.getProperties() method, 43-45
System.getProperty() method, 43-45, 466
System.in, 393
System.loadLibrary() method, 606
System.out, copying program output to, 592
System.out.print() method, 103
System.out.println() method, 6, 110, 286
System.Properties, 43-47
System.setErr() method, 398
System.setProperty method, 84

T

Tabs class, 133-137
tabs to spaces conversion, strings, 133-137
tailMap() method, 273
tailSet() method, 273
Tait, Andrew, 474
@Target annotation, 580, 581
TCP/IP connection, 496
TeamCity, 96
tee command, 399
TeePrintStream, 399-401
TempConverter program, 212-214
TemporalAccessors, 220
TemporalAdjusters factory class, 232
temporary/transient files, 411-414
TensorFlow, 472
terminal operation, 339
terminal, reading from, 392-396
ternary operator, 197
@Test annotation with JUnit, 78
test command (Maven), 63
Test-Driven Development (TDD), 77
testing
code for version or OS at runtime, 46
continuous integration, 95-100
with IDE, 12-18
unit testing for errors in code, 57, 77-81
TestNG, 80
text
line by line reading of files, 381-384
network transfer of, 501-503
reading or writing, 379-381
text blocks, 110, 128, 404
text formatter sample, 146
TFTP (Trivial File Transfer Protocol), 493
Thompson, Ken, 225
Thread class, 74, 424, 523
thread pool, 422, 425-426, 427, 430, 523
Thread.start() method, 424
Thread.stop() method, 424

threaded I/O, 380
threading, 421-454
  implementing, 423-429
  locks, 436-440
  parallel processing with Fork/Join, 447-450
  Queue interface, 444-446
  scheduling tasks, 451-454
  sharing data among threads, 440-444
  synchronizing, 432-437
  timeouts, 431-432
  virtual threads, 429-431
ThreadLocal class, 440-442, 444
time zones and computing dates, 234-235
time-zone classes, 218
timeouts, threading, 431-432
Timer service, 451-452
times (see dates and times)
TimeUnit class, 437
timing program, reflection, 583-585
TLS (Transport Layer Security), 529-532
toArray() method, 276
toBinaryString() method, 195
toCharArray() method, 124
today's date or time, 218-220
toHexString() method, 195
toInstant().getEpochSecond() method, 378
toJson() method, 541
tokenizing, 113-115
tolerance, floating-point number equality, 187
toLowerCase() method, 138
Tomcat, 526
toOctalString() method, 195
toString() method
  Arrays, 244
  Class, 555
  converting integers to alternative number base, 195
  converting numbers to strings, 178
  JSON with ChatGPT, 478
  JSONArray/JSONObject, 546
  object formatting, 286-288
  period between dates, 229
  platform-independent code, 404
  with StringBuilder, 111, 122
toUpperCase() method, 138
TR type parameter, 346
training, in machine learning, 469
transform() method (String), 109
transient/temporary files, 411-414
transitive dependencies, 62
Transport Layer Security (TLS), 529-532
TreeMap, 272-273
TreeSet, 261, 272-273
Tribuo, 473
triple slash (///), Markdown, 28
Trivial File Transfer Protocol (TFTP), 493
try-with-resources, socket connection, 497
try/catch block, 394
tryLock() method, 436
type parameter, 247
type parameter, and Collections, 250
type pattern matching, 311-313
typesafe, making enumerations, 308-309

U

UDP (user datagram protocol), 507-509
Unicode characters, 380
  converting to/from Strings, 129-131
  encoding a text file, 401
  Java regex package, 152
  operating systems' handling of, 131
UNICODE_CASE flag, 171, 172
unindenting strings, 128
unit testing, 57, 77-81, 81-82
Unix file permissions (see PosixFilePermission)
Unix Network Programming (Stevens), 512
Unix pipelines, and Streams, 329
Unix Power Tools (Peek, Powers, O'Reilly, and Loukides), 149
UNIX_LINES flag (regex), 172, 174
UnknownHostException, 498
unlock() method, 436
Unsafe class, 53
up call, 603
uptime command, 54
URI, URL, and URN, 510-511
URL object (ClassLoader), 410
URL, reading from web client, 494-496
URLConnection class, 494, 495
user datagram protocol (UDP), 507-509
UserMessage, 486
UTF-16, 130
utility classes, downloading, 7-12

V

var keyword, variable definition, 249
Vector arithmetic package, 203-207
Vector class, 204, 207, 250, 253

Vector.add(), 250
Vector.get(), 250
version and name, assigning, 60
version control systems, 11, 53
versions
  compatibility with book information, xiii
  dependency management with Maven, 70
  JUnit series, 80
  writing code that depends on Java version, 45
virtual threads, 422, 429-431, 525
VisualVM tool, 101
Vlissades, John, 77, 283
volatile modifier, variables in JNI, 606
VSCode IDE, 13, 17

W

Wadler, Philip, 252
wait() method, 444
waitFor() method, 595
walk() method, 369-418
walkFileTree() methods, 369-418
walking a file tree, 417-419
WAR (web archive) file, 37
Warburton, Richard, 330
warn() method, 86
WatchService class, 415
web application, 38
web page display of data, R, 467-468
web server, 526
web services, 494
web tier resources, packaging into single file, 37
when function (Mockito), 82
while loop, 259
who program (Unix), 399
WildFly (EE application server), 64, 534
windowFixed() method, 343
windowSliding() method, 343
with() method, 232
withLocale() method, 222
word frequency count algorithm, 341
words in a string, reversing, 132
WordUtils class, 139
work-stealing, 450
wrapper types, 177
writeLock() method, 437
Writer class, 359, 379-380, 383
writing files (see input/output)

X

Xcode, 17
XDoclet, 30
XMPP, 512
XP (Extreme Programming), 11, 77
XPath, 548

Y

yacc parser generator, 391

Z

Zeal (Mac documentation viewer), 30
Zip archives, 404-408
ZIP files, 404-408
ZipEntry class, 404-408
ZipFile class, 404-408
ZipFile.getInputStream() method, 404
ZipFile.getName() method, 404
ZipFile.stream() method, 405
ZipOutputStream class, 408
ZonedDateTime class, 234-235, 378

About the Author

Ian F. Darwin has worked in the computer industry for several decades. He wrote the freeware file(1) command used on Linux and BSD, and is the author of *Checking C Programs with Lint*, *Android Cookbook*, and more than a hundred articles and several courses on C, Unix, and Java at undergraduate level and commercially. In addition to programming and consulting, Ian teaches Unix and Java courses for Learning Tree International, one of the world's largest technical training companies. He was the first member of the Sun/Oracle "Java Champions" advocacy program. He has a M.Sc. in Computing from Staffordshire University and several technical certifications. His eclectic website can be found at *https://darwinsys.com*. Along with his wife and children, Ian used to raise chickens on their rural property north of Toronto.

Colophon

The animal on the cover of *Java Cookbook* is a Jersey hen, a breed of domestic chicken (*Gallus domesticus*). Chickens are descended from the wild red jungle fowl of Asia. Domesticated over 8,000 years ago, chickens are raised for meat and eggs.

With big, heavy bodies and small wings, chickens can fly only short distances. They are well suited to living on the ground. Their four-toed feet are designed for scratching in the dirt, where they find their usual diet: worms, bugs, seeds, and plants.

A male chicken is called a rooster and a female is known as a hen. The incubation period for a chicken egg is about three weeks; newly hatched chickens are precocial, meaning they have downy feathers and can walk around on their own right after emerging from the egg. They are not dependent on their mothers for food; not only can they procure their own, but they can live for up to a week after hatching on egg yolk that remains in their abdomen after birth.

The topic of chickens comes up frequently in ancient writings. Chinese documents date their introduction to China to 1400 BC, Babylonian carvings mention them in 600 BC, and Aristophanes wrote about them in 400 BC. The rooster has long symbolized courage: the Romans thought chickens were sacred to Mars, god of war, and the first French Republic chose the rooster as its emblem.

The IUCN does not assess the status of domesticated animals. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Dover. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.